CATalog Documentation

This is a document meant to explain how the CATalog application code works. It is also a way for me to do any necessary debugging before final publication.

One: Data Structure

The primary way that CATalog works is through a set of internal .csv files that are read by the application on startup. Two of these files are shown to the user (foreground.csv') the remaining files (background.csv, deltas.csv, and go_data.csv) are used internally by the application.

1-1: CSV Files

Each CSV file contains a different section of the database. CATalog was designed such that these files can be swapped out rather easily provided that they are structured similarly to the original dataset. A common trait shared between these files is the 'Entry' column. This column represents the 'proteins' and is used to connect different datasets.

1-1-1: Foreground

This is the primary file that is displayed to the user; it contains averaged and rounded intensity values for each protein relative to a given biofluid.

Entry	Protein	Gene	Urine	Plasma	Serum
A0A5F5XC	Alpha-1-B	A1BG	31.5	31.6	31.5
M3W0W4_	Alpha-1-B	A1BG	20	19.9	20.2
M3W3E7_F	Alpha-2-m	A2M PZP	24.3	25.7	25.6
A0A5F5Y1	Alpha-2-m	A2M PZP	26.6	30.6	30.6
A0A5F5Y3	Alpha-2-m	A2M PZP	17.5	19.4	19.7
A0A337S3	Alpha-2-m	A2ML1	22.3	26.7	26.8
M3WN23_	ATP bindin	ABCA6	19.1	11.6	11.3

1-1-2: Background

This file is not displayed. It contains full values for each sample; these values are averaged for the foreground table. Additionally, the values in this table are used to generate the boxplots.

Entry	Reviewed	Entry Nam	Protein na	Gene Nam	AU	BU	CU	DU	EU	HU	GU	FU	BP	AP
A0A5F5XC	unreviewe	A0A5F5XC	Alpha-1-B	A1BG	31.3022	31.01467	32.28593	31.29986	31.22713	31.23362	32.30432	31.57507	31.47495	31.2439
M3W0W4	unreviewe	M3W0W4_	Alpha-1-B	A1BG	19.8943	19.38823	20.47157	15.89057	15.84983	22.37332	23.09663	23.2305	15.4932	15.5589
M3W3E7	unreviewe	M3W3E7_F	Alpha-2-m	A2M PZP	25.25537	23.38098	25.26956	24.01522	24.58062	24.253	23.73027	23.66938	26.55741	25.27921
A0A5F5Y1	unreviewe	A0A5F5Y1	Alpha-2-m	A2M PZP	25.44843	27.13949	24.82691	25.06006	28.99808	25.67655	28.10796	27.87625	30.68382	30.12956
A0A5F5Y3	unreviewe	A0A5F5Y3	Alpha-2-m	A2M PZP	20.91451	19.99337	17.72418	17.59459	16.39557	19.23405	13.67135	14.19765	21.26607	22.26781
A0A337S3	unreviewe	A0A337S3	Alpha-2-m	A2ML1	22.35916	21.88088	21.09341	21.09989	24.83838	21.69763	22.90946	22.49598	26.58834	25.94046
M3WN23	unreviewe	M3WN23_	ATP bindin	ABCA6	20.19058	20.29625	20.83587	12.18237	16.16466	20.20884	21.37277	21.29095	11.23883	12.84915
A0A337S6	unreviewe	A0A337S6	ATP bindin	ABCB9	20.98336	22.12493	20.80871	20.81384	21.46013	21.87961	23.46564	23.24842	23.91505	22.03228
A0A337RX	unreviewe	A0A337RX	Putative pr	ABHD14B	25.50441	25.14283	25.0411	27.04927	25.90998	25.2098	25.96472	26.48494	22.01596	19.37539
A0A337SD	unreviewe	A0A337SD	ABI family	ABI3BP	21.42868	19.63318	20.13427	22.00841	20.50838	19.46856	18.77468	19.15224	19.60372	18.82175
A0A337S1	unreviewe	A0A337S1	Costars fa	ABRACL	17.85805	18.05548	17.423	18.40372	16.55828	17.51478	18.49716	17.76629	18.56669	20.30425
A0A337S9	unreviewe	A0A337S9	Medium-cl	ACADM	20.77087	18.52712	18.39883	20.03226	19.93334	17.34497	19.44975	17.79788	10.97247	10
M3W0B5	unreviewe	M3W0B5_I	Short/bran	ACADSB	21.06719	20.90814	20.04383	18.50297	22.33574	20.87347	25.6814	17.15607	19.55756	16.9004
A0A5F5XD	unreviewe	A0A5F5XD	Aggrecan c	ACAN	21.32503	21.15103	21.23596	20.16599	18.28199	17.91538	23.31459	20.47	18.921	15.28165

1-1-3: Deltas

This file is not displayed. It contains information pertaining to which biofluid is most intense for a given protein, using the average values contained in the foreground data. Each other average is compared to this maximum value and the average difference is reported. If the difference is less than one, the protein for that biofluid is marked as a '0', otherwise, it is marked as a '1'. These binary values are used to subset the foreground data by most intense biofluid as part of the filtering process.

Entry	Urine	Plasma	Serum	Max	DeltaUrine	DeltaSerur	DeltaPlasn	FlagUrine	FlagSerum	FlagPlasma
A0A5F5XC	31.5	31.6	31.5	31.6	0.1	0	0.1	0	0	0
M3W0W4_	20	19.9	20.2	20.2	0.2	0.3	0.2	0	0	0
M3W3E7_F	24.3	25.7	25.6	25.7	1.4	0	1.4	1	0	1
A0A5F5Y1	26.6	30.6	30.6	30.6	4	0	4	1	0	1
A0A5F5Y3	17.5	19.4	19.7	19.7	2.2	0.3	2.2	1	0	1
A0A337S3	22.3	26.7	26.8	26.8	4.5	0.1	4.5	1	0	1
M3WN23_	19.1	11.6	11.3	19.1	0	7.5	0	0	1	0
A0A337S6	21.8	22.8	18.3	22.8	1	0	1	1	0	1
A0A337RX	25.8	20.8	21	25.8	0	5	0	0	1	0
A0A337SD	20.1	19.2	18.6	20.1	0	0.9	0	0	0	0
A0A337S1	17.8	18.8	18.2	18.8	1	0	1	1	0	1

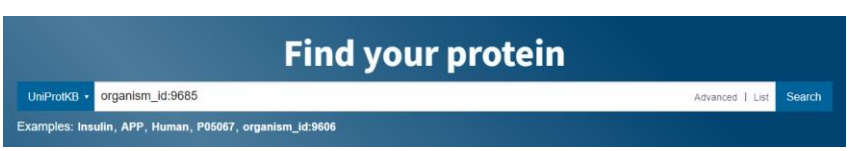
1-1-4: Go Data

The original version of this table isn't displayed, however, each cell that contains GO information is displayed as a generated dataframe.

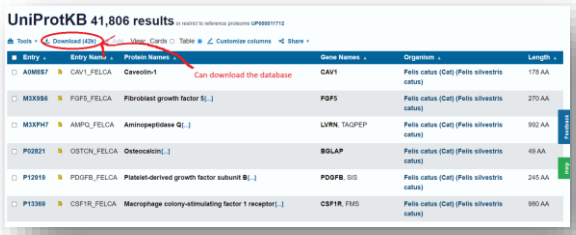
Column1	Column2	Column3	Column4	Column5	Column6	Column7
Entry	Entry Nam	Protein na	Gene Nam	Gene Onto	Gene Onto	Gene Ontology (molecular function)
A0A0A0MF	A0A0A0MF	Rhodopsin	RHO	absorption	cell-cell ju	G protein-coupled photoreceptor act
A0A0A0MF	A0A0A0MF	Cytochrom	CYP1A2	alkaloid m	endoplasm	aromatase activity [GO:0070330]; ca
A0A0A0MF	A0A0A0MF	C-C motif c	CCL5	antimicrob	cytoplasm	CCR chemokine receptor binding [GO
A0A0A0MF	A0A0A0MF	Amyloid-be	APP	adult locor	apical part	enzyme binding [GO:0019899]; hepa
A0A0A0MF	A0A0A0MF	Alkaline ph	ALPL	bone mine	extracellul	ADP phosphatase activity [GO:00432
A0A0A0MF	A0A0A0MF	Multifuncti	IL1B	cellular res	cytosol [G	cytokine activity [GO:0005125]; integ
A0A0A0MF	A0A0A0MF	Glyceralde	GAPDH	antimicrob	cytoskelet	aspartic-type endopeptidase inhibito

Which cell is displayed is controlled by the user through a set of radio buttons. More information on this can be found in chapter five.

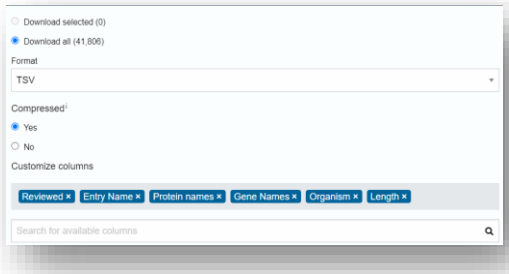
This data was originally obtained from the UniProt database for the taxon id 9685, which corresponds to the domestic cat.



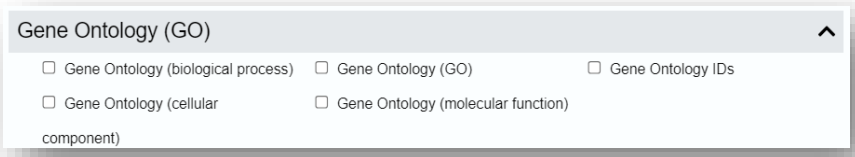
In the search results, there is an option to download the entire set of results in many different ways.



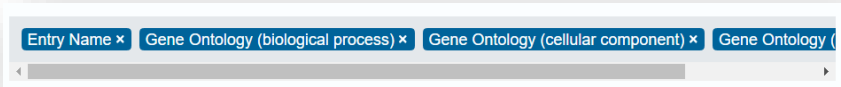
The default is to download as a FASTA file, however, the format can be easily changed to a TSV file which can be imported into excel.



We want the GO information which can be found in the 'UniProt data' section.



To build the final table, only the entry name and three GO categories were included:



1-2 Foreground Display:

This is the main data table featured in the application. It displays the foreground .csv file, which is contained in data$main. It also can be filtered by both Biofluid and associated GO terms.

UI code:

```
box(width = NULL, DT::dataTableOutput("display"),
    style = "height:400px; overflow-y: scroll; overflow-x: scroll;"),
```

Sever code:

```
  output$display <- DT::renderDataTable({
    datatable(main$data, selection = 'single')
  })
```

Interaction code:

```
 observeEvent(input$display_rows_selected,{
    main$row_selected <- input$display_rows_selected
    go_list <- go_chunk(data = main$data,
                        row_id = input$display_rows_selected,
                        GO_data
    )
    main$go_list <- go_list
    main$go_element <- fetch_go_info(go_list, field = input$go_item)
  })
```

The interaction code controls what happens when the user clicks on a row of the dataframe. 'main$row_selected' is the row number of the selected row. The purple portion of this code will be detailed in chapter five.

1-3: Application Layout

The application uses the ShinyDashboard package, which is a supplement to the existing shiny default layout and creates an overall cleaner look. Additional documentation on this package can be found at: https://rstudio.github.io/shinydashboard/. Complete code for the UI is as follows:

```
ui <- dashboardPage(skin = "black",
    dashboardHeader(title =
tags$img(src='https://i.ibb.co/x6tH34j/logo4.png', height = '60', width =
'120')),
    dashboardSidebar(
        textInput("keyword", "Filter proteins by GO: ", value = ""),
```

```
        actionButton("searchButton", "Search"),
        actionButton("resetButton", "reset"),
        selectInput("sampleType", "Filter by highest biofluid:",
                    choice = c("all", "urine", "serum", "plasma")),
        actionButton("filterButton", "Filter"),
        radioButtons("plot_labels", "Sample annotation: ",
                     c("off", "on")),
        radioButtons("go_item", "GO Data: ",
                     c("biological process",
                       "cellular compartment",
                       "molecular function"),
                     selected = "biological process")
    ),
      dashboardBody(
        fluidRow(
          column(width = 8,
            box(width = NULL, DT::dataTableOutput("display"),
              style = "height:400px; overflow-y: scroll; overflow-x:
scroll;"),
            box(width = NULL, DT::dataTableOutput("results"),
                style = "height: 200px; overflow-y: scroll; overflow-x:
scroll;")
            ),
          column(width = 4,
            box(width = NULL, plotOutput("boxplot", height = 300, width =
250)),
            box(width = NULL, div(tableOutput("demo"), style = "font-
size:70%; overflow-y: scroll"),
              style = "height: 100px")
          )
        )
    )
)
```

As a side note, the logo for the application is hosted on ImgBB in order to allow it to exist in the header.

Two: Searching

The search functionality is highly integrated with several other components of the application. The user enters a query word, the current GO database is searched, and the foreground data is filtered by proteins associated with that query. However, there is as a rather complex feature introduced to fix a strange interaction between searching and filtering.

To help bridge some gaps, there is a separate reactive container called ‘search’.

```
search <- reactiveValues()
```

The only value that this container holds is a boolean that indicates whether or not the foreground dataset is currently being truncated due to the query. A backup of this state is stored in the main container as a ‘search_cache’.

```
main$search_cache <- NULL
search$onging <- FALSE
```

It should be noted that the search function described in this section is not related to the search box present in the main data table. As described in the first section, that search box is part of the primary data table.

2-1: Search Box

The purpose of this element is to receive user input and store it as a string.

UI code:

```
textInput("keyword", "Filter proteins by GO: ", value = ""),
```

It is empty by default and the user text entry is stored as ‘input$keyword’. Additionally, there is no associated server code with this element.

2-2: Search Button

The search button is what actually uses the ‘input$keyword’ variable.

UI code:

```
actionButton("searchButton", "Search"),
```

Server code:

```
observeEvent(input$searchButton,{
    main$data <- foreground()
    main$back_data <- background()
    index <- go_column_mapper(input$go_item)
    res <- search_go_data(GO_data, main$data, index, word = input$keyword)
    main$data <- res
    res_background <- search_go_data_background(GO_data, main$back_data,
index, word = input$keyword)
    main$back_data <- res_background
    main$search_cache <- res
    print(nrow(main$search_cache))
    search$ongoing <- TRUE
  })
```

First, the table is reset and a 'res' dataframe is created by querying the foreground data for the keyword. This is both displayed by replacing the current foreground data in the main container (main$data) and stored in the cache as well. Additionally, the background data is similarly updated to ensure the plots remain consistent. Finally, the value 'search$ongoing' is set to true.

2-3: Reset Button

This simply restores the values back to defaults. The key here is to indicate that we are no longer executing a search.

UI code:

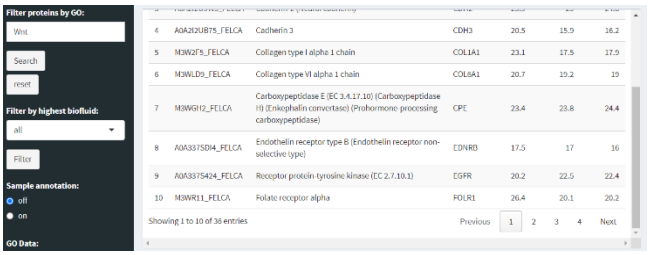
```
actionButton("resetButton", "reset"),
```

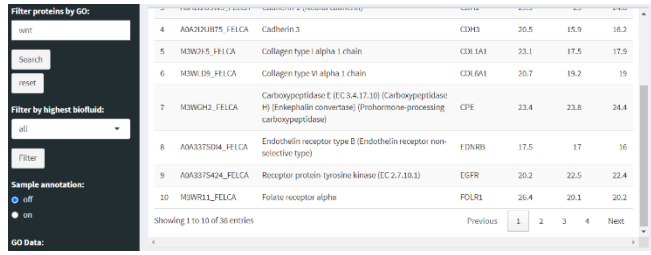
Server code:

```
observeEvent(input$resetButton,{
    main$data <- foreground()
    main$back_data <- background()
    updateTextInput(session,"keyword", value="")
    search$ongoing <- FALSE
  })
```

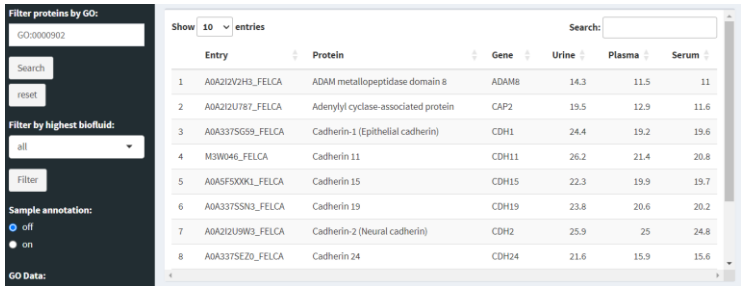
2-4: Search Example

If we search for ‘wnt’, ‘WNT’, or ‘Wnt’, the same results should be returned. The search is not case-sensitive and it matches partial words.





Further, searching by GO accession number is also possible. This includes and excludes brackets.



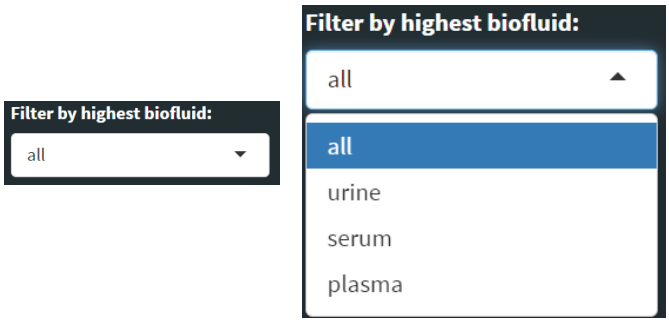
Three: Filtering

This portion has been giving me quite a lot of trouble. On the surface, it seemed like a rather simple task, however, there is a particularly insidious issue that needed to be ironed out. The main thing we are concerned with here is a static ‘deltas’ variable which is composed of a dataframe read from the ‘deltas.csv’ file.

```
deltas <- read.csv("deltas.csv")
```

3-1: Biofluid Selection

This is where the biofluid to select by is chosen. It is simple drop-down menu with a limited number of choices. The default choice is ‘all’.



UI Code:

```
        selectInput("sampleType", "Filter by highest biofluid:",
                    choice = c("all", "urine", "serum", "plasma")),
```

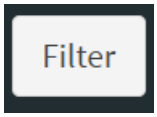
Sever code:

```
observeEvent(input$sampleType,{
    main$sample_selection <- input$sampleType
  })
```

All that is really happening is that the current selection is stored in the ‘main’ container as a string.

3-2: Filter Button

This is where things get complicated.



UI code:

```
        actionButton("filterButton", "Filter"),
```

Server code:

```
observeEvent(input$filterButton,{
    if(main$sample_selection == "all" & search$onging == FALSE){
      main$data <- foreground() #reset the table
    }
    else if(main$sample_selection == "all" & search$onging == TRUE){
      main$data <- main$search_cache #use cached search data
    }
    else if(main$sample_selection != "all" & search$ongoing == TRUE){
      main$data <- main$search_cache
      output <- filter_foreground_new(main$search_cache, deltas, field =
main$sample_selection)
      main$data <- output
    }
    else{
      main$data <- foreground()
      output <- filter_foreground_new(main$data, deltas, field =
main$sample_selection)
      main$data <- output
    }
  })
```

Most of the complexity is related to the fact that we have four possible different events here that can occur based on what the selected biofluid (main$sample_selection) and the search state (search$ongoing).

Condition 1: ‘all’ is selected and there is no ongoing search.

Resets the table.

Condition 2: ‘all’ is selected and there is an ongoing search.

Resets the table using the cached search results.

Condition 3: ‘all’ is not selected and there is an ongoing search.

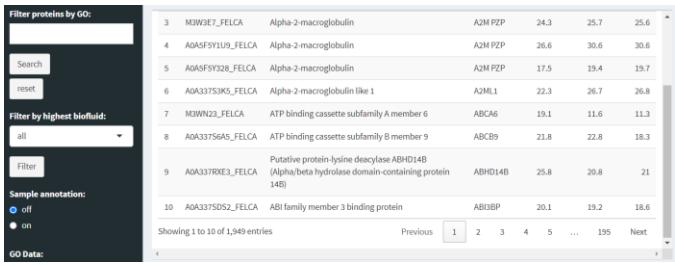
Filters the table using the cached search results.

Condition 4: ‘all’ is not selected and there is no ongoing search.

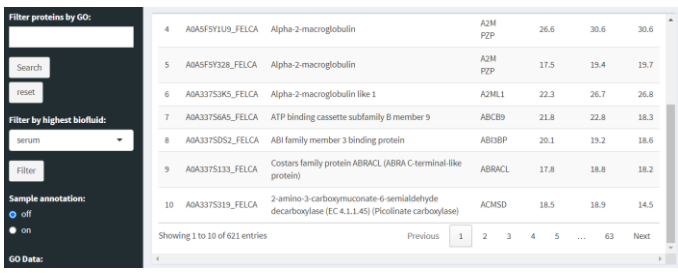
Resets the table, then filters the table.

3-3: Filtering Example

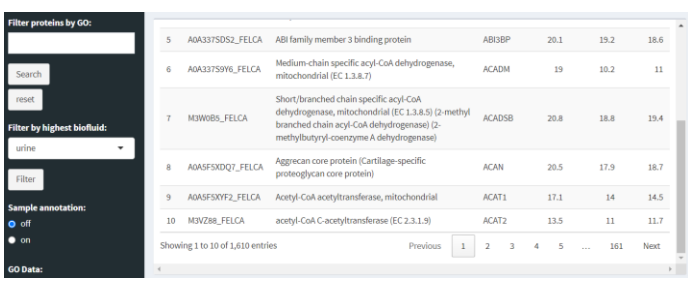
The entire database contains 1949 entries.



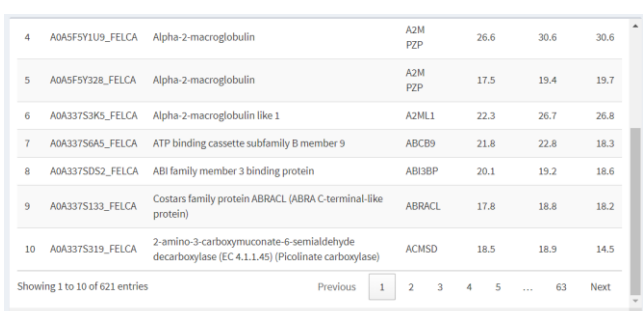
If we choose to filter by ‘serum’, this number drops to 621 entries.



Filtering by ‘urine’ gives 1610 entries.



If we filter by ‘serum’ again, the results are again 621 entries.



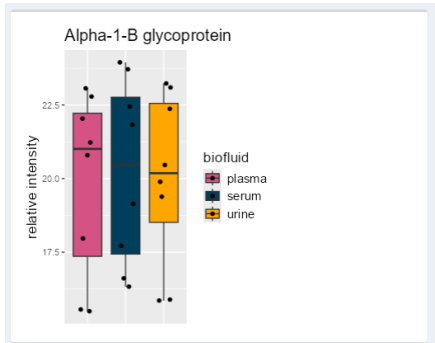
Because we have a rather robust set of conditions related to the ‘filter’ button, only one filter is applied at a time. For another example that uses the search cache, we can search for “golgi” as a query. This gives 17 search results.

If we filter by ‘serum’, these results are cut down to 4. Filtering by ‘urine’ gives 16 results.

Initiating a search before applying a filter causes the ‘search$ongoing’ variable to be set to TRUE, which tells the filter button to use the cached search results instead of resetting the table between subsequent filters. If we move back to ‘all’, we can see the original search results. The search cache will continue to be used until the reset button is pressed and ‘search$ongoing’ is set to FALSE again.

Four: Plots

The right side of the application displays boxplots for the selected row based on each sample in the background data.



A plot is made using ggplot 2 each time a row in the primary display is selected.

UI code:

```
box(width = NULL, plotOutput("boxplot", height = 300, width = 250)),
```

Server code:

```
output$boxplot <- renderPlot({
  req(input$display_rows_selected)
  s = input$display_rows_selected
  #use of a static background object here is a bit dangerous
  #this is what is causing the desynchronization between the plots and the search
  plot <- boxplot_wrapper(data = main$back_data, i = s, flag = main$plot_annotation)
  plot
 })
```

Use of a ‘req’ statement here prevents the application from trying to make a boxplot with empty data.

4-1: Annotations

Some additional user options were put into place regarding plot annotations. A flag is part of the primary boxplot_wrapper function that indicates whether or not to use text annotations instead of simple points. This flag is stored in ‘main$plot_annotation’ controlled by a radio button.

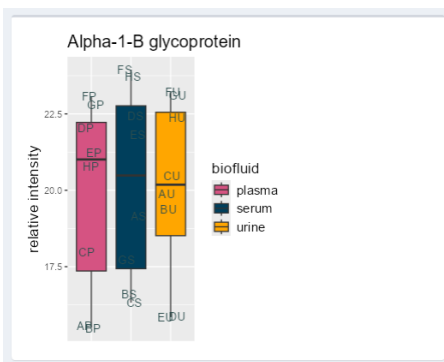
UI code:

```
radioButtons("plot_labels", "Sample annotation: ",
                     c("off", "on")),
```

Server code:

```
observeEvent(input$plot_labels,{
    main$plot_annotation <- input$plot_labels
    if(main$plot_annotation == "off"){
      main$demographics <- NULL
    }
    if(main$plot_annotation == "on"){
      main$demographics <- demographics
    }
  })
```

In addition to controlling the annotation of the plots themselves, these buttons also state whether or not the demographics box is displayed.



4-2: Demographics

This is a static table that sits underneath the boxplot and states the sex and age of each cat. It is only visible if the plot is currently being annotated, otherwise, it is stored as 'NULL'.

UI code:

```
box(width = NULL, div(tableOutput("demo"), style = "font-size:70%; overflow-y: scroll"),
              style = "height: 100px")
```

Server code:

```
  output$demo <- renderTable({
    main$demographics
```

```
})
```

Cat	A	B	C	D	E	F	G	H
Age	7	5	3	11	8	10	1	3
Sex	FS	MN	MN	MN	FS	FS	MN	FS

Five: GO Annotations

The go_data.csv file is a way of sorting each protein based on their associated gene ontology (GO) data in one of three categories: biological process, cellular compartment, and molecular function. This portion of the application is a bit difficult to explain without understanding the underlying data structure, so the following protein will be used as an example throughout this chapter.

10	A0A337SDS2_FELCA	ABI family member 3 binding protein	ABI3BP	20.1	19.2	18.6

5-1: GO Data Structure

The example protein is stored as a row in the 'go_data.csv' file with the following structure:

A0A337SD	A0A337SD	ABI family	ABI3BP	extracellular matrix organization [GO:0030198]	collagen-c	collagen binding [GO:0005518]; heparin binding [GO:0008201]

Although difficult to visualize here, each set of annotations is contained in a separate column and represents a 'cell':

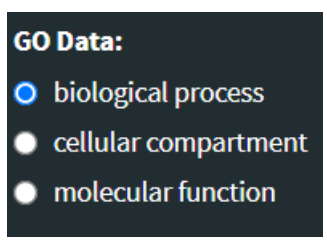
collagen binding [GO:0005518]; heparin binding [GO:0008201]

For instance, here is the entry for rhodopsin:

Column1	Column2	Column3	Column4	Column5	Column6	Column7
Entry	Entry Nam	Protein nai	Gene Nam	Gene Ontology (biological process)	Gene Ontology (cellular component)	Gene Ontology (molecular function)
A0A0A0MF	A0A0A0MF	Rhodopsin	RHO	absorption of visible light [GO:0016038]; cellular response to light stimu	cell-cell junction [GO:0005911]; Golgi ar	G protein-coupled photoreceptor activity [GO:0008020]; metal ion binding [GO:0046872]

5-2: GO Data Buttons

This set of buttons selects what column is shown in the main GO results table.



UI code:

```
radioButtons("go_item", "GO Data: ",
                     c("biological process",
                       "cellular compartment",
                       "molecular function"),
                     selected = "biological process")
```

Server code:

```
observeEvent(input$go_item,{
    main$go_element <- fetch_go_info(go_list = main$go_list, field =
input$go_item)
  })
```

By default, ‘biological process’ is selected. Which column is currently being used is stored in the ‘main$go_element’ variable.

5-3: GO Results Box

This is a data table box that shows a transformed version of the selected cell. The main idea is that the cell’s contents, which are stored as a string, are transformed into a two-column dataframe, where one column represents the GO term and the other represents the GO id.

collagen binding [GO:0005518]; heparin binding [GO:0008201]

Becomes:

	Description	GO_ID
1	collagen binding	GO:0005518
2	heparin binding	GO:0008201

Showing 1 to 2 of 2 entries Previous 1 Next

The above example is for the molecular function; selecting a different GO category changes what is displayed in this box. For example, here is the ‘cellular compartment’:

Show 10 entries Search:

	Description	GO_ID
1	collagen-containing extracellular matrix	GO:0062023
2	interstitial matrix	GO:0005614

Showing 1 to 2 of 2 entries Previous 1 Next

UI code:

```
box(width = NULL, DT::dataTableOutput("results"),
```

```
                style = "height: 200px; overflow-y: scroll; overflow-x:
scroll;")
            ),
```

Server code:

```
output$results <- DT::renderDataTable({
    req(input$display_rows_selected)
    main$go_element
  })
```

What is being displayed here is ‘main$go_element’. The actual dataframe is not stored as a file, rather, it is repeatedly generated and destroyed in response to user input. Specifically, it is created as part of the ‘input$go_item’ buttons. Again, a ‘req’ statement is used to prevent display issues.

Six: Functions

Highlighted in green were functions not related to Shiny. These are what actually process the data and were written by me. For the purposes of documentation, I've grouped them together by what they're related to.

6-1: Boxplot Functions

The underlying code for the boxplot deals heavily with ggplot2, which is a great tool for easily customizing plots.

6-1-1: Boxplot Wrapper

This is the primary function that holds boxplots together. It uses a bit of branching logic to determine whether or not the plot should be annotated or unannotated.

Inputs:

- Data: background dataset
- i: index of the selected row
- flag: whether or not plot annotations are enabled

Returns: a ggplot2 object

```
boxplot_wrapper <- function(data, i, flag){
	output <- format_data(data, i)
	name <- unlist(output[1])
	df <- as.data.frame(output[[2]])
	if(flag == "off"){
		plot <- make_boxplot_unannotated(df, name)
	}
	if(flag == "on"){
		plot <- make_boxplot_annotated(df, name)
	}
	plot
}
```

6-1-2: Format Data

Generates a dataframe based on a given row of the background data that can be made into a boxplot.

Inputs:

- data: background dataset
- i: index of the selected row

Returns: a list containing the protein name as well as the dataframe to be made into a boxplot.

```
format_data <- function(data, i){
      x <- data[i,]
      name <- x[,4]
      x <- subset(x, select = -c(Entry, Reviewed, Entry.Name, Protein.names,
Gene.Names))
      values <- as.numeric(x)
      labels <- colnames(x)

      urine <- replicate(8, "urine")
      plasma<- replicate(8, "plasma")
      serum <- replicate(8, "serum")

      biofluid <- c(urine, plasma, serum)

      df <- data.frame(values, biofluid, labels)

      output <- list(name, df)
      output
}
```

6-1-3: Make Boxplot Annotated

Creates a boxplot with text annotations.

Inputs:

- Data: background dataset
- Name: name of the selected protein

Returns: a ggplot2 object

```
make_boxplot_annotated <- function(df, name){
      plot <- ggplot(df, aes(x = as.factor(biofluid), y = values, fill =
biofluid, label = labels))+
            geom_boxplot(outlier.shape = NA)+
```

```
            theme(axis.title.x=element_blank(),
                  axis.text.x=element_blank(),
                  axis.ticks.x=element_blank(),
                  axis.title.y = element_text(size = 14),
                  plot.title = element_text(size = 16),
                  legend.title=element_text(size = 14),
                  legend.text=element_text(size = 12))+
            scale_fill_manual(values = c('#D55382', '#003F5C', '#FFA600'))+
            geom_text(color = "darkslategrey", position =
position_jitter(seed = 1, width = 0.2))+
            ylab("relative intensity")+
            ggtitle(name)
      plot
}
```

6-1-4: Make Boxplot Unannotated

Creates a boxplot with points instead of annotations.

Inputs:

- Data: background dataset
- Name: name of the selected protein

Returns: a ggplot2 object

```
make_boxplot_unannotated <- function(df, name){
      plot <- ggplot(df, aes(x = as.factor(biofluid), y = values, fill =
biofluid, label = labels))+
            geom_boxplot(outlier.shape = NA)+
            theme(axis.title.x=element_blank(),
                  axis.text.x=element_blank(),
                  axis.ticks.x=element_blank(),
                  axis.title.y = element_text(size = 14),
                  plot.title = element_text(size = 16),
                  legend.title=element_text(size = 14),
                  legend.text=element_text(size = 12))+
            scale_fill_manual(values = c('#D55382', '#003F5C', '#FFA600'))+
            geom_jitter(color = "black", position = position_jitter(seed = 1,
width = 0.2))+
```

```
            ylab("relative intensity")+
            ggtitle(name)
      plot
}
```

6-2: GO Functions

These are related to parsing the GO dataset. This includes both setting up the GO annotation display as well as searching proteins related to a specific GO term.

6-2-1: GO Chunk

Wrapper function that gets a list of GO annotations.

Inputs:

- Data: foreground data
- row_id: index of the selected row
- go_data: go database

Returns: a list of Go annotations. This is a list of dataframes, each one corresponds to a different GO category.

```
go_chunk <- function(data, row_id, go_data){
      protein <- go_protein_mapper(data, row_id)
      go_list <- cell_parser_wrapper(protein, go_data)
      go_list
}
```

6-2-2: GO Protein Mapper

Extracts the Uniprot entry ID for a selected row.

Inputs:

- data: foreground data
- row_id: index of the selected row

Returns: Uniprot entry id for selected protein.

```
go_protein_mapper <- function(foreground, i){
      r <- foreground[i,]
      protein <- r$Entry
```

```
	protein
}
```

6-2-3: Cell Parser Wrapper

Generates the list of annotation dataframes for a given protein.

Inputs:

- id: Uniprot entry id for the selected protein.
- data: GO database

Returns: list of GO annotation dataframes.

```
cell_parser_wrapper <- function(id, data){
	bio_process <- parse_cell(id, 5, data)
	cell_comp <- parse_cell(id, 6, data)
	mol_func <- parse_cell(id, 7, data)
	go_list <- list(bio_process, cell_comp, mol_func)
	go_list
}
```

6-2-4: Parse Cell

Formats a selected cell in the GO annotation table.

Inputs:

- protein: Uniprot entry id for the selected protein.
- Index: column to parse.
- Data: GO database

Returns: GO annotation dataframe

```
parse_cell <- function(protein, index, data){
	r <- data%>%
		filter(Column2 == protein)
	info <- r[,index]
	semi_parsed <- unlist(strsplit(info, split = ';'))
	df <- data.frame(semi_parsed)%>%
		tidyr::separate(semi_parsed, into = c("Description", "GO_ID"),
sep = "\\[|\\]", extra = "drop", fill = "right")
	df
```

```
}
```

6-2-5: Fetch GO Info

Retrieves a specific dataframe from the list of GO annotation dataframes.

Inputs:

- Go_list: list of GO annotation dataframes.
- Field: type of GO information to retrieve; this is directly connected to the user input.

Returns: selected dataframe.

```
fetch_go_info <- function(go_list, field){
	print(field)
	if(field == "biological process"){ #default
		output <- go_list[[1]]
	}
	if(field == "cellular compartment"){
		output <- go_list[[2]]
	}
	if(field == "molecular function"){
		output <- go_list[[3]]
	}
	output
}
```

6-2-6: Search GO Data Background / Search GO Data Foreground

Finds all proteins whose GO annotation matches a keyword for a selected category, then filters the background data based on this information. These two functions do essentially the same thing, the only difference is the data being searched. The purpose of searching the background data, even though it is not displayed, is to ensure that the correct boxplot is displayed when a row in the foreground data is selected as the selection process uses numeric row indices instead of entry names.

Inputs:

- Data: go database
- Background: background dataset
- Index: GO category to search; this corresponds to the input from the radio button.
- Word: keyword to search; this is direct user input.

Returns: filtered dataframe.

```
search_go_data_background <- function(data, background, index, word){
	df <- data[grepl(word, data[,index], ignore.case = TRUE),]
	entries <- df$Column2
	res <- background%>%
		filter(Entry.Name %in% entries)
	res
}
```

6-2-7: GO Column Mapper

Connects the user selected category (string) to a numeric index that can be used to retrieve a given GO category column to search using the previous function.

Inputs:

- Field: GO category go parse; direct input from the radio buttons.

Returns: column index that should be searched.

```
go_column_mapper <- function(field){
	index <- 5
	if(field == "biological process"){
		index = 5
	}
	if(field == "cellular compartment"){
		index = 6
	}
	if(field == "molecular function"){
		index = 7
	}
	index
}
```

6-3: Other Functions

These functions are related to loading the data or other data processing.

6-3-1: Load Foreground Data / Load Background Data

These functions load the foreground dataset and background dataset respectively. They exist to set up these datasets as reactive objects.

Inputs:

- None

Returns: a dataframe.

```
load_background_data <- function(){
	data <- read.csv("./data/background.csv")
	data
}
```

6-3-2: Filter Foreground

This function simply returns every element whose delta is less than 1 (indicated by a '0') for a given biofluid.

Inputs:

- Data: foreground dataset
- Deltas: delta dataset
- Field: selected biofluid

Returns: a truncated version of the foreground data which contains proteins that are most intense in the selected biofluid.

```
filter_foreground <- function(data, deltas, field){
	if(field == "urine"){
		res <- deltas%>%
			filter(FlagUrine == 0)
	}
	if(field == "serum"){
		res <- deltas%>%
			filter(FlagSerum == 0)
	}
	if(field == "plasma"){
		res <- deltas%>%
			filter(FlagPlasma == 0)
	}
```

```
    entries <- res$Entry
    output <- data%>%
          filter(Entry %in% entries)

}
```