# Package 'systemPipeShiny'

February 27, 2021

**Title** systemPipeShiny: An Interactive Framework for Workflow Management and Visualization

**Version** 1.0.2

**Date** 2021-01-14

**Description** systemPipeShiny (SPS) extends the widely used systemPipeR (SPR) workflow environment with a versatile graphical user interface provided by a Shiny App. This allows non-R users, such as experimentalists, to run many systemPipeR's workflow designs, control, and visualization functionalities interactively without requiring knowledge of R. Most importantly, SPS has been designed as a general purpose framework for interacting with other R packages in an intuitive manner. Like most Shiny Apps, SPS can be used on both local computers as well as centralized server-based deployments that can be accessed remotely as a public web service for using SPR's functionalities with community and/or private data. The framework can integrate many core packages from the R/Bioconductor ecosystem. Examples of SPS' current functionalities include: (a) interactive creation of experimental designs and metadata using an easy to use tabular editor or file uploader; (b) visualization of workflow topologies combined with auto-generation of R Markdown preview for interactively designed workflows; (d) access to a wide range of data processing routines; (e) and an extendable set of visualization functionalities. Complex visual results can be managed on a 'Canvas Workbench' allowing users to organize and to compare plots in an efficient manner combined with a session snapshot feature to continue work at a later time. The present suite of preconfigured visualization examples. The modular design of SPR makes it easy to design custom functions without any knowledge of Shiny, as well as extending the environment in the future with contributions from the community.

**Depends** R (>= 4.0.0), shiny (>= 1.5.0), shinyTree

**Imports** DOT, DT, R6, RSQLite, assertthat, colourpicker, crayon, dplyr, ggplot2, glue, markdown, magrittr, methods, networkD3, openssl, plotly, rhandsontable, rlang, rstudioapi, shinyAce, shinyFiles, shinyWidgets, shinydashboard, shinydashboardPlus, shinyjqui, shinyjs, shinytoastr, stringr, stats, styler, utils, vroom (>= 1.3.1), yaml

**Suggests** testthat, BiocStyle, systemPipeR, knitr, rmarkdown

**VignetteBuilder** knitr

**biocViews** Infrastructure, DataImport, Sequencing, QualityControl, ReportWriting, ExperimentalDesign, Clustering

**License** Artistic-2.0

**Encoding** UTF-8

**LazyData** true

**BugReports** https://github.com/systemPipeR/systemPipeShiny/issues

**URL** https://github.com/systemPipeR/systemPipeShiny

**RoxygenNote** 7.1.1

**Roxygen** list(markdown = TRUE)

**Config/testthat/edition** 3

**git_url** https://git.bioconductor.org/packages/systemPipeShiny

**git_branch** RELEASE_3_12

**git_last_commit** b4a3bdc

**git_last_commit_date** 2021-01-12

**Date/Publication** 2021-02-26

**Author** Le Zhang [aut, cre],
Daniela Cassol [aut],
Ponmathi Ramasamy [aut],
Jianhai Zhang [aut],
Gordon Mosher [aut],
Thomas Girke [aut]

**Maintainer** Le Zhang <le.zhang001@email.ucr.edu>

# R topics documented:

---

`addData` *Add and get data between SPS tabs*

---

## Description

These functions are designed to be used inside SPS tabs

## Usage

```
addData(data, shared, tab_id)

getData(tab_id, shared)
```

## Arguments

| | |
|---|---|
| `data` | any type of R object you want to store and use in other tabs |
| `shared` | the a `shared` reactivevalues object that is defined on the top level server. Read vignette for more details about this object |
| `tab_id` | tab ID of current tab if using `addData` method and tab ID to get data from if using `getData`. |

## Value

Nothing to return with `add` method and returns original object for the `get` method

## Examples

```
if(interactive()){
    spsInit()
    options(sps = list(verbose = TRUE))
    ui <- fluidPage(
        useToastr(),
        actionButton("add", "add"),
        actionButton("get", "get"),
        actionButton("wrong", "when it gets wrong")
    )
    server <- function(input, output, session) {
        shared <- reactiveValues()
        data <- dplyr::tibble(this = 123)
        cat('before adding\n')
        print(shared)
```

```
        observeEvent(input$add, {
            addData(data, shared, "core_about")
            cat('after adding\n')
            print(shared) # watch the data_intask object is created
        })
        observeEvent(input$get, {
            cat("get data\n")
            print(getData('core_about', shared))
        })
        observeEvent(input$wrong, {
            cat("get wrong data\n")
            getData('not_there', shared)
        })
    }
    shinyApp(ui, server)
}
```

---

`clearableTextInput` *A clearable text inputInput control*

---

## Description

An UI component with a "X" button in the end to clear the entire entered text. It works the same as `Textinput`.

## Usage

```
clearableTextInput(inputId, label, value = "", placeholder = "")
```

## Arguments

| | |
|---|---|
| `inputId` | ID |
| `label` | text label above |
| `value` | default value |
| `placeholder` | place holder text |

## Details

must be call `useSps()` at head to load css and js

## Value

a div

## Examples

```
if(interactive()){

    ui <- fluidPage(
        useSps(),
        clearableTextInput("input1", "This is a input box")
    )
```

```
    server <- function(input, output, session) {

    }

    shinyApp(ui, server)
}
```

---

`dynamicFile` *Dynamically generate Shiny file selection component*

---

## Description

Depending on the "mode" in SPS options, this function renders a similar UI components but behaves differently on server. `local` mode will not copy file, directly use a path pointer, `server` mode upload file and store in temp. Expect similar behavior as shiny::fileInput.

## Usage

```
dynamicFile(
  id,
  title = "Select your file:",
  label = "Browse",
  icon = NULL,
  style = "",
  multiple = FALSE
)
```

## Arguments

| | |
|---|---|
| `id` | element ID |
| `title` | element title |
| `label` | upload button label |
| `icon` | button icon, only works for `local` mode |
| `style` | additional button style, only works for `local` mode |
| `multiple` | multiple files allowed |

## Value

a Shiny upload component

## Examples

```
if(interactive()){
    library(shinyjs)
    # change to 'local' to see the difference
    options(sps = list(mode='server'))
    ui <- fluidPage(
        useShinyjs(),
        dynamicFile("getFile"),
        textOutput("txt_file")
    )
```

```
    server <- function(input,output,session){
        runjs('$(".sps-file input").attr("readonly", true)')
        myfile <- dynamicFileServer(input,session, id = "getFile")
        observe({
            print(myfile())
        })
    }
    shinyApp(ui = ui, server = server)
}
```

---

`dynamicFileServer` *Server side function for dynamicFile*

---

## Description

Server side function for dynamicFile to parse the uploaded file path

## Usage

```
dynamicFileServer(input, session, id)
```

## Arguments

| | |
|---|---|
| `input` | shiny server input |
| `session` | shiny server session |
| `id` | input file element ID. Do not us `ns()` to wrap the id on server side if inside module |

## Value

reactive dataframe, need to extract the value inside reactive expression, observe, or inside `isolate`

## Examples

```
if(interactive()){
    library(shinyjs)
    options(sps = list(mode='server')) # Change the mode to 'local' to see difference
    ui <- fluidPage(
        useShinyjs(),
        dynamicFile("getFile"),
        textOutput("txt_file")
    )

    server <- function(input,output,session){
        runjs('$(".sps-file input").attr("readonly", true)')
        myfile <- dynamicFileServer(input, session, id = "getFile")
        observe({
            print(myfile()) # remember to use `()` for reactive value
        })
    }
    shinyApp(ui = ui, server = server)
}
```

## emptyIsFalse *Empty objects and FALSE will return FALSE*

### Description

judge if an object is empty or FALSE, and return FALSE if it is

### Usage

```
emptyIsFalse(x)
```

### Arguments

| | |
|---|---|
| x | any R object |

### Details

not working on S4 class objects.

Useful for if statement. Normal empty object in if will spawn error. Wrap the expression with `emptyIsFalse` can avoid this. See examples

### Value

`NA`, `""`, `NULL`, `length(0)`, `nchar == 0` and `FALSE` will return `FALSE`, otherwise `TRUE`.

### Examples

```
emptyIsFalse(NULL)
emptyIsFalse(NA)
emptyIsFalse("")
try(`if(NULL) "not empty" else "empty"`) # will generate error
if(emptyIsFalse(NULL)) "not empty" else "empty" # this will work
# similar for `NA`, `""`, `character(0)` and more
```

## gallery *A shiny gallery component*

### Description

`texts`, `hrefs`, `images` Must have the same length

### Usage

```
gallery(
  Id = NULL,
  title = "Gallery",
  title_color = "#0275d8",
  texts,
  hrefs,
  images,
  image_frame_size = 4
)
```

## Arguments

| | |
|---|---|
| `Id` | ID of this gallery |
| `title` | Title of gallery |
| `title_color` | Title color |
| `texts` | label under each image |
| `hrefs` | link when clicking each |
| `images` | image source, |
| `image_frame_size` | integer, 1-12, this controls width |

## Details

`useSps()` must be called in UI before using this function.

The version usually been used in SPS framework is `genGallery()`

## Value

a div element

## Examples

```
if(interactive()){
    texts <- c("p1", "p2", "p3", "p4", "p5")
    hrefs <- c("https://unsplash.it/1200/768.jpg?image=251",
               "https://unsplash.it/1200/768.jpg?image=252",
               "https://unsplash.it/1200/768.jpg?image=253",
               "https://unsplash.it/1200/768.jpg?image=254",
               "https://unsplash.it/1200/768.jpg?image=255")
    images <- c("https://unsplash.it/600.jpg?image=251",
                "https://unsplash.it/600.jpg?image=252",
                "https://unsplash.it/600.jpg?image=253",
                "https://unsplash.it/600.jpg?image=254",
                "https://unsplash.it/600.jpg?image=255")
    library(shiny)

    ui <- fluidPage(
        useSps(),
        gallery(texts = texts, hrefs = hrefs, images = images)
    )

    server <- function(input, output, session) {

    }

    shinyApp(ui, server)
}
```

genGallery *Generate gallery by only providing tab names*

## Description

A fast way in SPS to generate a gallery to display plot tab screenshots

## Usage

```
genGallery(
  tab_ids = NULL,
  Id = NULL,
  title = "Gallery",
  type = NULL,
  title_color = "#0275d8",
  image_frame_size = 3,
  app_path = NULL
)
```

## Arguments

| | |
|---|---|
| tab_ids | a vector of tab IDs |
| Id | element ID |
| title | gallery title |
| type | If this value is not NULL, filter by tab type, and tab_ids will be ignored. One of c("core", "wf", "data", "vs"). use spsTabInfo() to see tab information |
| title_color | title color, common colors or hex code |
| image_frame_size | integer, 1-12 |
| app_path | app path, default current working directory |

## Details

require a SPS project and the config/tabs.csv file. If you want to use gallery outside a SPS project, use gallery()

## Value

gallery div

## Examples

```
if(interactive()){
    spsInit()
    ui <- fluidPage(
        genGallery(c("plot_example1")),
        genGallery(type = "plot")
    )
    server <- function(input, output, session) {

    }
```

```
    shinyApp(ui, server)
}
```

---

genHrefTab *Display a list of links in a row of buttons*

---

## Description

hrefTab can be use for any purpose of shiny. genHrefTab is upper level wrapper of hrefTab and should only be used under SPS framework for fast retrieving tab info and generate the hrefTab. To use genHrefTab, the tab_info.csv config file must be there in config directory. label_text, hrefs must be the same length

## Usage

```
genHrefTab(
  tab_ids,
  Id = NULL,
  title = "A bar to list tabs",
  text_color = "#0275d8",
  app_path = NULL,
  ...
)

hrefTab(
  Id = NULL,
  title = "A list of tabs",
  title_color = "#0275d8",
  label_text,
  hrefs,
  ...
)
```

## Arguments

| | |
|---|---|
| tab_ids | tab names, must have the tab_info dataframe |
| Id | optional element ID |
| title | Item title |
| text_color | Table text color |
| app_path | app path, default is current working directory |
| ... | other arguments to be passed to the html element |
| title_color | title color |
| label_text | individual tab labels |
| hrefs | individual tab links |

## Details

*genHrefTab* require a SPS project and the config/tabs.csv file. If you want to use hrefTab outside a SPS project just use *hrefTab*

## Value

a HTML element

## Examples

```
if(interactive()){
    ui <- fluidPage(
        useSps(),
        hrefTab(label_text = c("Bar Plot", "PCA Plot", "Scatter Plot"),
                hrefs = c("https://google.com/", "", ""))
    )

    server <- function(input, output, session) {

    }
    shinyApp(ui, server)
}
```

---

genHrefTable *Generate a table that lists tabs by rows*

---

## Description

A fast way in SPS to generate a table that lists SPS tabs

## Usage

```
genHrefTable(
  rows,
  Id = NULL,
  title = "A Table to list tabs",
  text_color = "#0275d8",
  app_path = NULL,
  ...
)
```

## Arguments

| | |
|---|---|
| rows | a named list of character vector, the item names in the list will be the row names and each item should be a vector of tab IDs. Or you can use one of 'core', 'wf', 'vs', 'data', 'plot' to specify a tab type, so it will find all tabs matching that type. See `tab_info.csv` under `config` directory for type info. |
| Id | element ID |
| title | table title |
| text_color | text color for table |
| app_path | app path, default is current working directory |
| ... | any additional arguments to the html element, like class, style... |

## Details

For rows, there are some specially reserved characters for type and sub-types, one of c('core', 'wf', 'vs', 'data', 'plot'). If indicated, it will return a list of tabs matching the indicated tabs instead of searching individual tab names. See examples.

This function requires a SPS project and the config/tabs.csv file. If you want to use hrefTable outside a SPS project, or want to create some links pointing to outside web resources, use hrefTable

## Value

HTML elements

## Examples

```
if(interactive()){
    spsInit()
    # will be two rows, one row is searched by tab IDs and the other is
    # searched by type.
    rows <- list(row1 = c("core_canvas", "core_about"),
                 row2 =  "data")
    ui <- fluidPage(
        genHrefTable(rows)
    )
    server <- function(input, output, session) {

    }
    shinyApp(ui, server)
}
```

---

# hexLogo *Hexagon logo and logo panel*

---

## Description

Shiny UI widgets to generate hexagon logo(s). hexLogo() generates a single hexagon, and hexPanel() generates a panel of hex logos

## Usage

```
hexLogo(
  id,
  title = "",
  hex_img,
  hex_link = "",
  footer = "",
  footer_link = "",
  x = "-10",
  y = "-20"
)

hexPanel(
  id,
  title,
```

```
  hex_imgs,
  hex_links = NULL,
  hex_titles = NULL,
  footers = NULL,
  footer_links = NULL,
  xs = NULL,
  ys = NULL
)
```

## Arguments

| | |
|---|---|
| id | input ID |
| title | title of the logo, display on top of logo or title of logo panel displayed on the left |
| hex_img | single value of hex_imgs |
| hex_link | single value of hex_links |
| footer | single value of footers |
| footer_link | single value of footer_links |
| x | character of number, X offset, e.g. "-10" instead of -10L |
| y | character of number, Y offset |
| hex_imgs | a character vector of logo image source, can be online or local, see details |
| hex_links | a character vector of links attached to each logo, if not NULL, must be the same length as hex_imgs |
| hex_titles | similar to hex_links, titles of each logo |
| footers | a character vector of footer attached to each logo |
| footer_links | a character vector of footer links, if not NULL, must be the same length as footers |
| xs | a character vector X coordinate offset value for each logo image, default -10 |
| ys | Y coordinates offset, must be the same length as xs, default -20 |

## Details

The image in each hexagon is resized to the same size as the hex border and then enlarged 125%. You may want to use x, y offset value to change the image position.

If your image source is local, you need to add your local directory to the shiny server, e.g. `addResourcePath("sps","ww` This example add www folder under my current working directory as sps to the server. Then you can access my images by `hex_imgs = "sps/my_img.png"`.

some args in hexPanel are character vectors, use NULL for the default value. If you want to change value but not all of your logos, use "" to occupy space in the vector. e.g. I have 3 logos, but I only want to add 2 footer and only 1 footer has a link: `footers = c("footer1","footer2","")`, `footer_links = c("","https://mylink","")`. By doing so footers and footer_links has the same required length.

useSps() must be called on Shiny UI to load the required css style.

## Value

HTML elements

## Examples

```
if(interactive()){
    ui <- fluidPage(
        useSps(),
        hexPanel(
            "demo1", "DEMO 1:" ,
            rep("https://live.staticflickr.com/7875/46106952034_954b8775fa_b.jpg", 2)
        ),
        hexPanel(
            "demo2", "DEMO 2:" ,
            rep("https://live.staticflickr.com/7875/46106952034_954b8775fa_b.jpg", 2),
            rep("https://www.google.com", 2),
                 c("hex1", "hex2")
        ),
        hexPanel(
            "demo3", "DEMO 3:" ,
            rep("https://live.staticflickr.com/7875/46106952034_954b8775fa_b.jpg", 2),
            footers = c("hex1", "hex2"),
            footer_links = rep("https://www.google.com", 2)
        )
    )
    server <- function(input, output, session) {
    }
    shinyApp(ui, server)
}
```

---

`hrefTable` *A table of hyper reference buttons*

---

## Description

creates a table in Shiny which the cells are hyper reference buttons

## Usage

```
hrefTable(
  Id = NULL,
  title = "A Table of list of tabs",
  text_color = "#0275d8",
  item_titles,
  item_labels,
  item_hrefs,
  ...
)
```

## Arguments

| | |
|---|---|
| `Id` | optional |
| `title` | title of this table |
| `text_color` | text color |
| `item_titles` | vector of strings, a vector of titles for table row names |

| | |
|---|---|
| item_labels | list, a list of character vectors to specify button labels in each table item |
| item_hrefs | list, a list of character vectors to specify button hrefs links |
| ... | other HTML args |

## Details

item_titles, item_labels, item_hrefs must have the same length. Each item in item_labels, item_hrefs must also have the same length. For example, if we want to make a table of two rows, the first row has 1 cell and the second row has 2 cells:

```
hrefTable(
    item_titles = c("row 1", "row 2"),
    item_labels = list(c("cell 1"), c("cell 1", "cell 2")),
    item_hrefs = list(c("link1"), c("link1", "link2")
)
```

Must call useSps() in UI.

The more often used versuin in SPS framework is genHrefTable()

## Value

HTML elements

## Examples

```
if(interactive()){

    ui <- fluidPage(
        useSps(),
        hrefTable(item_titles = c("workflow 1", "workflow 2"),
                  item_labels = list(c("tab 1"), c("tab 3", "tab 4")),
                  item_hrefs = list(c("https://www.google.com/"), c("", "")),
        )

    )

    server <- function(input, output, session) {

    }

    shinyApp(ui, server)
}
```

---

# loadDF *Load tabular files as tibbles to server*

---

## Description

load a file to server end. It's designed to be used with the input file source switch button(see it in a SPS new template data tab). It uses vroom::vroom to load the file. In SPS, this function is usually combined as downstream of dynamicFileServer() function on on the server side to read the file into R. This loading function only works for parsing tabular data, use vroom::vroom() internally.

## Usage

```
loadDF(
  choice,
  data_init = NULL,
  upload_path = NULL,
  eg_path = NULL,
  comment = "#",
  delim = "\t",
  col_types = vroom::cols(),
  ...
)
```

## Arguments

| | |
|---|---|
| choice | where this file comes from, from 'upload' or example 'eg'? |
| data_init | a tibble to return if upload_path or eg_path is not provided. Return a 8x8 empty tibble if not provided |
| upload_path | when choice is "upload", where to load the file, will return data_init if this param is not provided |
| eg_path | when choice is "eg", where to load the file, will return data_init if this param is not provided |
| comment | comment characters when load the file, see help file of vroom |
| delim | delimiter characters when load the file, see help file of vroom |
| col_types | columns specifications, see help file of vroom |
| ... | other params for vroom, see help file of vroom |

## Details

This is function is wrapped by the shinyCatch() function, so it will show loading information both on console and on UI. This function prevents errors to crash the Shiny app, so any kind of file upload will not crash the app. To show message on UI, useSps() must be used in Shiny UI function, see examples.

## Value

returns a tibble or NULL if parsing is unsuccessful

## Examples

```
if(interactive()){
    library(shinyWidgets)
    # change value to 'local' to see the difference
    spsOption("mode", value = "server")
    ui <- fluidPage(
        useSps(),
        shinyWidgets::radioGroupButtons(
            inputId = "data_source", label = "Choose your data file source:",
            selected = "upload",
            choiceNames = c("Upload", "Example"),
            choiceValues = c("upload", "eg")
        ),
        dynamicFile("data_path", label = "input file"),
```

```
        dataTableOutput("df")
    )

    server <- function(input, output, session) {
        tmp_file <- tempfile(fileext = ".csv")
        write.csv(iris, file = tmp_file)
        upload_path <- dynamicFileServer(input, session, "data_path")
        data_df <- reactive({
            loadDF(choice = input$data_source,
                   upload_path = upload_path()$datapath,
                   delim = ",", eg_path = tmp_file)
        })
        output$df <- renderDataTable(data_df())
    }
    shinyApp(ui, server)
}
```

---

`makePlotData` *Create data receiving methods for plot tabs*

---

## Description

This function specify for each input data type in a plot tab,

1. where the data is coming from, 2. how to validate incoming data. To use this function, make sure there is a SPS project and *config/tabs.csv* exists.

## Usage

```
makePlotData(
  dataset_id = "data",
  dataset_label = "Raw data",
  receive_datatab_ids = "data_example",
  vd_expr = spsValidate({    if (is.data.frame(mydata$data))        TRUE    else
    stop("Data xx needs to be a dataframe or tibble") }),
  app_path = getwd(),
  use_string = FALSE
)
```

## Arguments

| | |
|---|---|
| `dataset_id` | string, length 1, a unique ID within this plot tab. |
| `dataset_label` | string, length 1, what label to display on UI for this type of input data |
| `receive_datatab_ids` | a vector of tab IDs: for this kind of data input, which data tabs that can be used as input source(s). For example, if this plot tab requires a dataframe and can be produced from "data_df1" or "data_df2", *receive_datatab_ids = c("data_df1", "data_df2")*. These options are later rendered as a drop down menu for users to choose where they have prepared the required data from. |
| `vd_expr` | what expression to validate(check) the incoming data set. Usually it is a spsValidate object |
| `app_path` | path, SPS project folder |

`use_string` bool, if you don't want to parse *vd_expr*, use quoted string for *vd_expr* and turn this to TRUE. See the same argument in newTabPlot

## Details

For the validation expression, the incoming data is stored in a reactive values object, and you can access this data object by *mydata$dataset_id*, e.g. the dataset_id is "raw_data", then when the time you validate this type of incoming data set, a variable *mydata$raw_data* is accessible. So you can directly use `mydata$raw_data` in *vd_expr*.

It is recommended to create data tabs first before running this function, because *receive_datatab_ids* required data tab id exists in the *tabs.csv* file.

## Value

a special list that stores one type of data input info

## See Also

newTabPlot

## Examples

```
spsInit(change_wd = FALSE, overwrite = TRUE, project_name = "SPS_plotdata")
newTabData("data_df1", "df 1",
           app_path = "SPS_plotdata",
           open_file = FALSE)
newTabData("data_df2", "df 2",
           app_path = "SPS_plotdata",
           open_file = FALSE)
plotdata_raw <- makePlotData("raw", "raw data",
             receive_datatab_ids = "data_df1",
             vd_expr = spsValidate({
                 if(!is.data.frame(mydata$raw))
                     stop("Input raw data need to be a dataframe")
             }, vd_name = "Validate raw data"),
             app_path = "SPS_plotdata")
plotdata_meta <- makePlotData("meta", "meta data",
             receive_datatab_ids = c("data_df1", "data_df2"),
             vd_expr = spsValidate({
                 if(!is.data.frame(mydata$meta))
                     stop("Input raw data need to be a dataframe")
                 if(nrow(mydata$meta) < 1)
                     stop("Input raw data need to have at least one row")
             }, vd_name = "Validate meta data"),
             app_path = "SPS_plotdata")
newTabPlot("plot_test1",
           app_path = "SPS_plotdata",
           plot_data = list(plotdata_raw, plotdata_meta))
```

`makePrepro` *Create data tab preprocess methods*

## Description

On a data tab, given the same uploaded data set, users can choose different ways to preprocess the data and therefore different preprocessing methods will lead to different plot tab options. Every call of this function defines a preprocess method: 1. data validation expression before preprocess; 2. actual preprocess expression; 3. plot options after preprocess.

## Usage

```
makePrepro(
  method_id = "md1",
  label = "New method1",
  vd_expr = spsValidate(is.data.frame(data_filtered)),
  pre_expr = {     data_filtered },
  plot_options = "default",
  use_string = FALSE
)
```

## Arguments

| | |
|---|---|
| `method_id` | string, length 1, a unique ID within this data tab. |
| `label` | string, length 1, what label to display on UI for users to choose as a preprocess option |
| `vd_expr` | expression, usually a spsValidate object. Before preprocess if there is any additional validation that is special to this preprocess method, you can specify here |
| `pre_expr` | The actual preprocess expression. You should use a pre-created variable called *data_filtered* to start and this is the object that contains filtered data after users filtering on the UI. In the end of this expression, you should return a preprocessed dataframe or whatever object type that can be accepted by the desired plot tab. It is recommended to write the preprocess method into a function and directly call the function here, e.g. |

```
myPreprocess <- function(data){
    if(is.numeric(data[ ,1]))
        data[ ,1] <- data[ ,1] + 1
    return(data)
}
makePrepro(
 ...,
 pre_expr = myPreprocess(data_filtered),
 ...
)
```

| | |
|---|---|
| `plot_options` | plot tab IDs: if data is preprocessed by this method, what kind of plots it can make, specify plot tab IDs in a vector. Note: unlike the *receive_datatab_ids* argument in makePlotData that requires the *config/tabs.csv* exists, this argument doesn't require the config file or the plot tab to be existing. You can use any ID(s) here. The ID checking is postponed when the genGallery function runs |

on app start. "default" means all possible plot tabs, the same as *type = 'plot'* in genGallery.

`use_string` same as the argument in newTabPlot, controls *vd_expr* and *pre_expr* in this function

## Details

for *vd_expr*, *pre_expr* a variable called *data_filtered* is accessible and is the object where data stored. One should use this object to do validation or preprocess. See examples.

## Value

a special list that contains all info for a preprocess method

## Examples

```
spsInit(change_wd = FALSE, overwrite = TRUE, project_name = "SPS_prepro")
# first preprocess method
prepro_log <- makePrepro(
    "log", "take log of first column",
    vd_expr = spsValidate({
        if(!is.data.frame(data_filtered))
            stop("Input input data need to be a dataframe")
    }, vd_name = "log method pre-checks"),
    pre_expr = {
        if(is.numeric(data_filtered[ ,1]))
            {if(all(data_filtered[ ,1] > 0)){
                data_filtered[ ,1] <- log(data_filtered[ ,1])}
            }
        data_filtered
    },
    plot_options = c("plot_xx1", "plot_xx2")
    )
## remember to save these helper functions in a R scripts under the R
## folder. They will be automatically sourced when app starts.
myPreprocess <- function(data){
    if(is.numeric(data[ ,1]))
        data[ ,1] <- data[ ,1] + 1
    return(data)
}
myVd <- function(data, vd_name){
    spsValidate({
        if(!is.data.frame(data))
            stop("Input input data need to be a dataframe")
    }, vd_name = vd_name)
}
# second preprocess method
prepro_addone <- makePrepro(
    "addone", "add one to first column",
    vd_expr = myVd(data_filtered, "add one method pre-checks"),
    pre_expr = myPreprocess(data_filtered),
    plot_options = c("plot_xx1")
)
# Combine two methods and make a new data tab
newTabData("data_test1",  "test 1",
           app_path = "SPS_prepro",
```

```
  prepro_methods = list(prepro_log, prepro_addone)
  )
```

---

`msg` *SPS terminal message logging methods*

---

## Description

If SPS `use_crayon`option is `TRUE`, the message will be colorful. "INFO" level spawns `message`, "WARNING" is `warning`, "ERROR" spawns `stop`, other levels use `cat`. `spsinfo`, `spswarn`, `spserror` are higher level wrappers of `msg`. The only difference is they have SPS- prefix.

`spsinfo` has an additional arg `verbose`. This arg works similar to all other `verbose` args in this package, if not specified, it follows the project option, can be be forced to `TRUE` and `FALSE`. `TRUE` will forcefully generate the msg, and `FALSE` will be no message.

## Usage

```
msg(
  msg,
  level = "INFO",
  .other_color = "white",
  info_text = "INFO",
  warning_text = "WARNING",
  error_text = "ERROR"
)

spsinfo(msg, verbose = NULL)

spswarn(msg)

spserror(msg)
```

## Arguments

| | |
|---|---|
| `msg` | a character string of message or a vector of character strings, each item in the vector presents one line of words |
| `level` | typically, one of "INFO", "WARNING", "ERROR", lower case OK. Other levels ok. |
| `.other_color` | hex color code or named colors, when levels are not in "INFO", "WARNING", "ERROR", this value will be used |
| `info_text` | info level text prefix |
| `warning_text` | warning level text prefix |
| `error_text` | error level text prefix |
| `verbose` | bool, default get from sps project options, can be overwritten |

## Details

If you want to use this function to generate colorful log messages but not in SPS framework. Use spsOption to set up the option on very top of your scripts: `spsOption("use_crayon",TRUE)`.

## Value

see description

## Examples

```
msg("this is info")
msg("this is warning", "warning")
try(msg("this is error", "error"))
msg("this is other", "error2")
spsinfo("some msg, verbose false", verbose = FALSE) # will not show up
spsinfo("some msg, verbose true", verbose = TRUE)
spswarn("sps warning")
try(spserror("sps error"))
```

---

# newTabPlot *Create a new SPS tab*

---

## Description

Functions to create a new SPS tab. It is recommended to create the data tab first then its linked plot tabs.

## Usage

```
newTabPlot(
  tab_id = "plot_id1",
  tab_displayname = "Plot Tab Title",
  desc = "default",
  img = "",
  plot_expr = plotly::ggplotly(ggplot2::ggplot(mydata$data,
    ggplot2::aes_string(names(mydata$data)[1], names(mydata$data)[2])) +
    ggplot2::geom_point(ggplot2::aes(color = seq_len(nrow(mydata$data))))),
  pkgs = list(cran_pkg = c("base"), bioc_pkg = c(""), github = c("")),
  plot_data = list(makePlotData(app_path = app_path)),
  plot_out_func = plotly::plotlyOutput,
  plot_render_func = plotly::renderPlotly,
  app_path = getwd(),
  out_folder_path = file.path(app_path, "R"),
  plot_control_ui = tagList(h3("Some plotting options")),
  author = "",
  plugin = "",
  empty = FALSE,
  preview = FALSE,
  use_string = FALSE,
  reformat = TRUE,
  open_file = TRUE,
  verbose = spsOption("verbose"),
  colorful = spsOption("use_crayon")
)

newTabData(
```

```
  tab_id = "data_id1",
  tab_displayname = "Data Tab Title",
  desc = "default",
  pkgs = list(cran_pkg = c("base"), bioc_pkg = c(""), github = c("")),
  common_validation = spsValidate({     "pass" }, "common"),
  prepro_methods = list(makePrepro("nothing", "do nothing"), makePrepro("md1",
    "method1", vd_expr = {     nrow(data_filtered) > 1 })),
  app_path = getwd(),
  out_folder_path = file.path(app_path, "R"),
  eg_path = file.path(app_path, "data", "iris.csv"),
  plugin = "",
  author = "",
  empty = FALSE,
  preview = FALSE,
  reformat = TRUE,
  open_file = TRUE,
  use_string = FALSE,
  verbose = spsOption("verbose"),
  colorful = spsOption("use_crayon")
)
```

## Arguments

| | |
|---|---|
| tab_id | character string, length 1, must start with "*plot_*" for plot tabs and "*data_*" for data tabs. Must be a unique value. use spsTabInfo(app_path = "YOUR_APP_PATH") to see current tab IDs. |
| tab_displayname | character string, length 1, the name to be displayed on side navigation bar list and tab title |
| desc | character string, length 1 in markdown format. Tab description and instructions. You can make type it in multiple lines but in only one string (one pair of quotes). e.g. |

```
"
# some desc
## second line,
- bullet 1
- bullet 2
"
```

| | |
|---|---|
| img | realtive path, ideally a plot screenshot of users expect to see when they make the plot. It can be a internet link or a local link which uses the *www* folder asthe root. e.g. drop your image *plot.png* inside *www/plot_list*, then the link here is "plot_list/plot.png". Only needed for plot tabs. |
| plot_expr | the plot expression, like all other expression in other shiny reactive expressions. e.g for more than one line use . Only to be used for plot tabs. default is |

```
plotly::ggplotly(
    ggplot2::ggplot(mydata$data,
           ggplot2::aes_string(names(mydata$data)[1],
                     names(mydata$data)[2])
    ) +
    ggplot2::geom_point(aes(
            color = seq_len(nrow(mydata$data))
```

```
    ))
)
```

pkgs — which packages you require users to install, list. specify CRAN, bioconductor or github packages in a vector. see shinyCheckPkg

```
list(
    cran_pkg = c("base"),
    bioc_pkg = c(""),
    github = c("")
)
```

plot_data — a list of *makePlotData()* results, see makePlotData for details.

```
list(
    makePlotData("plot_1", ...),
    makePlotData("plot_2", ...),
    ...
)
```

plot_out_func — the plot output function to use on UI, like shiny::plotOutput, just the function name without quotes and without *()*. default is plotly::plotlyOutput.

plot_render_func — The render plot function to use on server. No quotes, no *()*. Must be **paired** with *plot_out_func*. e.g If plotly::plotlyOutput is used on UI, server must use plotly::renderPlotly. If you use shiny::renderPlot, plot will not show up.

app_path — string, app directory, default is current directory

out_folder_path — string, which directory to write the new tab file, default is the *R* folder in the SPS project

plot_control_ui — additional UI components you want to add to control plotting options, like additional slider bar, some on/off switches, text input etc. If more than one components, put them in a shiny::tagList

author — character string, or a vector of strings. authors of the tab

plugin — character string, if you are building a tab for a plugin, you can specify the plugin name here.

empty — bool, for **advanced developers**, if you don't want to use SPS default tab UI and server structure, you can use turn this to *TRUE*. A very simple template will be generated and you need to write all UI and server parts by yourself. In this case, only `tab_id`, `display_name`, `author` are needed, other tab args can be ignored, system args are still working, like `verbose`, `preview`, `style`, `colorful`

preview — bool, *TRUE* will print the new tab code to console and will not write the file and will not register the tab

use_string — bool, sometimes parsing an expression in R may not be totally accurate. To aviod this problem, turn this to *TRUE* and for *plot_control_ui*, *plot_expr*, *p_out_func*, *p_render_func*, wrap your expression in a quoted string. What you have provided in the string will be what on the new tab file, no expression parsing will happen. Can only be controlled as a group, which means use string for all of them or none of them in plot tabs. For data tab, the affected argument is *common_validation* . When turn this to *TRUE*, be careful with quotes in your expression, escape or use alternative of single/double quotes.

```
newTabPlot(
    ...
    use_string = TRUE,
    plot_control_ui = "
   tagList(clearableTextInput('id1', 'label')), h5('this title')
    ",
    plot_expr = "
    plotly::ggplotly(
        ggplot2::ggplot(mydata$data,
               ggplot2::aes_string(names(mydata$data)[1],
                           names(mydata$data)[2])) +
            ggplot2::geom_point(aes(
                color = seq_len(nrow(mydata$data))
            ))
    )
    ",
    p_out_func = "plotly::plotlyOutput"
    ...
)
```

`reformat` bool, whether to use styler::style_file reformat the code

`open_file` bool, if Rstudio is detected, open the new tab file?

`verbose` bool, default follows the project verbosity level. *TRUE* will give you more information on progress and debugging

`colorful` bool, whether the message will be colorful or not

`common_validation` expression, use '{}' to wrap around multiple line expressions. Usually a spsValidate object. You can use shiny's built in shiny::req or shiny::validate for a simpler version.

`prepro_methods` a list of makePrepro method returns, read help for that function for details

`eg_path` example data set path. Each data tab requires an example data set to be displayed when users don't have anything to upload. Usually this data file is a tabular file and stored in the *data* folder in a SPS project

## Details

- Must use this function inside a SPS project, use `spsInit()` if there is no project.
- For a new data tab, different preprocessing methods, their pre-prequirements and what plotting options available after each preprocess is controlled by the `makePrepro()` function. Each call from makePrepro function specify one preprocessing method. All preprocess methods should be provided in a list and passed to the *prepro_methods* argument.
- A new plot tab can have more than one data set as the input. For example a plot can require a metadata table and a log transformed table as inputs. There can also be multiple data tabs can preprocess and produce the same log table. So you need to specify how many data inputs this plot requires; for each input which data tab(s) this plot tab can receive data from; for each input data type, what validations (data format checks) you want to do. All of these are controlled by makePlotData and return(s) of this function should be provided in a list to the *plot_data* argument.
- Different reprocess methods in a data tab is controlled by the makePrepro function. Each call from this function specify one preprocessing method. All preprocess methods should be provided in a list to the *prepro_methods* argument.

- One of the steps in creating a plot tab is to specify incoming data source. This is controlled by `receive_datatab_ids` argument in `makePlotData()`. It requires the data tab IDs exist in the config file *config/tabs.csv*. So, it is best to create all required data tabs first. Or specify it to any existing data tab like 'data_example' and when the template is created, manually change it.

## Value

a tab file in R folder and tab info registered on config/tabs.csv

## Examples

```
spsInit(change_wd = FALSE, overwrite = TRUE)
newTabData(
    tab_id = "data_new",
    tab_displayname = "my first data tab",
    prepro_methods = list(makePrepro(label = "do nothing",
                                     plot_options = "plot_new")),
    app_path = glue::glue("SPS_{format(Sys.time(), '%Y%m%d')}")
)
newTabPlot(
    tab_id = "plot_new1",
    tab_displayname = "my first plot tab",
    plot_data = list(
        makePlotData(dataset_label = "Data from my new tab",
                     receive_datatab_ids = "data_new",
                     app_path = glue::glue("SPS_{format(Sys.time(), '%Y%m%d')}"))
    ),
    app_path = glue::glue("SPS_{format(Sys.time(), '%Y%m%d')}")
)
newTabData(
    tab_id = "data_empty",
    tab_displayname = "my first empty data tab",
    empty = TRUE,
    app_path = glue::glue("SPS_{format(Sys.time(), '%Y%m%d')}")
)
newTabPlot(
    tab_id = "plot_empty",
    tab_displayname = "my first empty plot tab",
    empty = TRUE,
    app_path = glue::glue("SPS_{format(Sys.time(), '%Y%m%d')}")
)
```

---

`pgPaneUpdate` *A draggable progress panel*

---

## Description

Creates a panel that displays multiple progress items. Use `pgPaneUI` on UI side and use `pgPaneUpdate` to update it. The UI only renders correctly inside `shinydashboard::dashboardPage()` or `shinydashboardPlus::dash`

A overall progress is automatically calculated on the bottom.

## Usage

```
pgPaneUpdate(pane_id, pg_id, value, session = getDefaultReactiveDomain())

pgPaneUI(pane_id, titles, pg_ids, title_main = NULL)
```

## Arguments

| | |
|---|---|
| pane_id | Progress panel main ID, use ns wrap it on pgPaneUI but not on pgPaneUpdate if using shiny module |
| pg_id | a character string of ID indicating which progress within this panel you want to update. Do not use ns(pg_id) to wrap it on server |
| value | 0-100 number to update the progress you use pg_id to choose |
| session | current shiny session |
| titles | labels to display for each progress, must have the same length as pg_ids |
| pg_ids | a character vector of IDs for each progress. Don't forget to use ns wrap each ID. |
| title_main | If not specified and pane_id contains 'plot', title will be 'Plot Prepare'; has 'df' will be 'Data Prepare', if neither will be "Progress" |

## Value

HTML elements

## Examples

```
if(interactive()){
    library(shinydashboard)
    # try to slide c under 0
    ui <- dashboardPage(header = dashboardHeader(),
                        sidebar = dashboardSidebar(),
                        body = dashboardBody(
                            useSps(),
                            h4("you need to open up the progress
                                tracker, it is collapsed ->"),
                            actionButton("a", "a"),
                            actionButton("b", "b"),
                            sliderInput("c", min = -100,
                                        max = 100, value = 0,
                                        label = "c"),
                            pgPaneUI("thispg",
                                     c("this a", "this b", " this c"),
                                     c("a", "b", "c"), "Example Progress")
                        )
    )
    server <- function(input, output, session) {
        observeEvent(input$a, {
            for(i in 1:10){
                pgPaneUpdate("thispg", "a", i*10)
                Sys.sleep(0.3)
            }
        })
        observeEvent(input$b, {
            for(i in 1:10){
```

```
                pgPaneUpdate("thispg", "b", i*10)
                Sys.sleep(0.3)
            }
        })
        observeEvent(input$c, pgPaneUpdate("thispg", "c", input$c))
    }
    shinyApp(ui, server)
}
```

---

`plotContainer` *SPS snapshots container*

---

## Description

Initiate this container at the global level, this is done for you in *global.R* and you only need to call the `sps_plots` object when you need methods from this class.

This container is used to communicate plotting tabs with the canvas tab.

## Public fields

`plot_ui` a list of plot UI snapshots will be stored here. You shouldn't manually edit this list, use `add/getUI` method

`plot_server` a list of plot server snapshots will be stored here. You shouldn't manually edit this list, use `add/getServer` method

## Methods

### Public methods:

- `plotContainer$new()`
- `plotContainer$addUI()`
- `plotContainer$getUI()`
- `plotContainer$addServer()`
- `plotContainer$getServer()`
- `plotContainer$notifySnap()`
- `plotContainer$clone()`

**Method** `new()`**:** initialize a new class container

*Usage:*

```
plotContainer$new()
```

**Method** `addUI()`**:** add plot UI to the container. use it to wrap around the original plot out put function

*Usage:*

```
plotContainer$addUI(plot_DOM, tab_id)
```

*Arguments:*

`plot_DOM` Plotting output function, like `plotOutput`

`tab_id` unique ID, usually use in a module and use the tab ID

**Method** `getUI()`**:** Get plot UI to the container.

*Usage:*

```
plotContainer$getUI(tab_id, plot_id_new = NULL)
```

*Arguments:*

`tab_id` unique ID, usually the tab ID if used in a module

`plot_id_new` string, usually if taking a snapshot of a plot DOM, they can't use the same ID on HTML. When get the UI, change the ID to a new value

*Returns:* a saved plotting DOM

**Method** `addServer()`**:** add plot server function to the container. use it to wrap around the original plot output server function, like `renderPlot`, `renderPlotly`

*Usage:*

```
plotContainer$addServer(
  render_func,
  tab_id,
  expr,
  env = parent.frame(),
  quoted = FALSE,
  ...
)
```

*Arguments:*

`render_func` plot output server function

`tab_id` unique ID, usually the tab ID if used in a SPS tab

`expr` the reactive expression to render the plot, same as `expr` in other render function

`env` default `parent.frame()`, see shiny documents

`quoted` Is you `expr` quoted?

`...` additional args to pass to the original render function

**Method** `getServer()`**:** Get plot UI to the container.

*Usage:*

```
plotContainer$getServer(tab_id)
```

*Arguments:*

`tab_id` unique ID, usually the tab ID if used in a module

*Returns:* saved plot server function

**Method** `notifySnap()`**:** notify the snapshot tab a snapshot has been added

*Usage:*

```
plotContainer$notifySnap(tab_id, skip = 1, reset = FALSE, set_to = NULL)
```

*Arguments:*

`tab_id` unique ID, usually the tab ID if used in a module

`skip` positive integer, This function is usually bound to the plotting button. When clicked, the first number of `skip` clicks will not add the snapshot to the canvas tab.

`reset` bool, to reset the count and return nothing

`set_to` positive integer, reset the count of a tab to be a certain number

*Returns:* a vector of `tab_id` and count number as a character string

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
plotContainer$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
# a simple example to show how the snapshots communicates with Canvas
if(interactive()){
    library(shiny)
    library(shinydashboard)
    library(shinyjs)
    plots = plotContainer$new()
    mod1_UI <- function(id) {
        ns <- NS(id)
        tagList(
            h1("a tiny example of how Canvas work in SPS"),
            actionButton(ns("render"), "render"),
            jqui_resizable(plots$addUI(plotlyOutput(ns("plot1")), id)),
            sliderInput(ns("slide"), label = "rows",
                        min = 1, max = nrow(iris),
                        value = nrow(iris))

        )
    }
    mod1 <- function(input, output, session, shared) {
        observeEvent(input$render, {
            output$plot1 <- plots$addServer(renderPlotly, 'mod1', {
                plotly::ggplotly(ggplot2::ggplot(iris[1:input$slide, ],
                                 ggplot2::aes(Sepal.Length, Sepal.Width)) +
                               ggplot2::geom_point(aes(colour = Species)))
            })
            shared$snap_signal <- plots$notifySnap("mod1")
            req(shared$snap_signal)
            toastr_info(
                glue("Snapshot {glue_collapse(shared$snap_signal, '-')}",
                     "added to canvas"),
                        position = "bottom-right")
        })
    }
    mod2_UI <- function(id){
        ns <- NS(id)
        tagList(
            actionButton(ns("refresh"), 'refresh'),
            sliderTextInput(
                inputId = ns("ncols"),
                label = "Number of columns per row to initiate canvas:",
                choices = c(1:4, 12), selected = 2, grid = TRUE
            ),
            fluidRow(uiOutput(ns("new")), class = "sps-canvas")
        )
    }

    mod2 <- function(input, output, session, shared) {
        ns <- session$ns
        make_plots <- reactiveValues(ui = list(), server = list())
        observeEvent(shared$snap_signal, {
            tab_id <- shared$snap_signal[1]
            new_plot_id <- glue("{tab_id}-{shared$snap_signal[2]}")
            print(new_plot_id)
            make_plots$ui[[new_plot_id]] <-
                list(plots$getUI(tab_id, ns(new_plot_id)), ns(new_plot_id))
```

```
    make_plots$server[[new_plot_id]] <- plots$getServer(tab_id)
})
observeEvent(input$refresh, {
    ui <- make_plots$ui
    output$new <- renderUI({
        sapply(seq_along(ui), function(i){
            column(
                width = 12/isolate(input$ncols),
                class = "collapse in",
                id = glue("{ui[[i]][2]}-container"),
                div(class = "snap-drag bg-primary",
                    h4(glue("Plot {ui[[i]][2]}")),
                    tags$button(
                        class = "btn action-button canvas-close",
                        icon("times"),
                        `data-toggle`="collapse",
                        `data-target`=glue("#{ui[[i]][2]}-container"),
                        `plot-toggle` = ui[[i]][2]
                    )
                ),
                jqui_resizable(make_plots$ui[[i]][[1]]),
                tags$script(glue(.open = '@', .close = '@',
                                 '$("#@ui[[i]][2]@-container")',
                                 '.draggable({ handle: ".snap-drag"})'))
            )
        }, simplify = FALSE) %>%{
            fluidRow(id = ns('plots'),
                     tags$head(tags$style(
                         '
                     .snap-drag {
                     opacity: 0;
                     };'),
                         tags$style(
                             '
                     .snap-drag button{
                     position: absolute;
                     outline: none;
                     background-color: Transparent;
                     top: 2px;
                     right: 4px;
                     };'),
                         tags$style(
                             '
                     .snap-drag h4{
                     margin-bottom: 0;
                     padding-bottom: 10px;
                     };'),
                         tags$style(
                             '
                     .snap-drag button:focus {
                     outline: 0 !important;
                     box-shadow: none !important;
                     };'),
                         tags$style('.snap-drag:hover {
                     opacity: 1;
                     }
                     '
```

```
                                )),
                            tagList(.)
                    )
                }
            })
            for(plot_id in names(make_plots$server)){
                output[[plot_id]] <- make_plots$server[[plot_id]]
            }
        })
        observeEvent(input$hide_title, {
            shinyjs::toggleClass(selector = ".snap-drag", class = "collapse")
        })

    }
    ui <- dashboardPagePlus(
        header = dashboardHeaderPlus(),
        sidebar = dashboardSidebar(),
        body = dashboardBody(
            useShinyjs(),
            useSps(),
            mod1_UI("mod1"),
            mod2_UI("mod2")),
        title = "Test",
    )
    server <- function(input, output, session) {
        shared = reactiveValues()
        callModule(mod1, "mod1",  shared)
        callModule(mod2, "mod2",  shared)
    }
    shinyApp(ui, server)
}
```

---

`quiet` *Suppress cat print output*

---

## Description

Useful if you want to suppress cat or print

## Usage

```
quiet(x)
```

## Arguments

| | |
|---|---|
| `x` | function or expression or value assignment expression |

## Value

If your original fucntions has a return, it will return in `invisible(x)`

## Examples

```
quiet(print(1))
quiet(cat(1))
```

`removeSpsTab` *Remove a SPS tab*

## Description

Remove a tab R file and remove from the tabs.csv config file

## Usage

```
removeSpsTab(
  tab_id = "none",
  force = FALSE,
  app_path = getwd(),
  multiple = FALSE,
  verbose = spsOption("verbose"),
  colorful = spsOption("use_crayon")
)
```

## Arguments

| | |
|---|---|
| `tab_id` | tab ID, string, length 1, supports regular expressions, so be careful. If more than one tabs are matched, stop by default |
| `force` | bool, whether to ask for confirmation |
| `app_path` | app directory |
| `multiple` | bool, if matched more than one tab, turn this to *TRUE* can remove more than one tab at a time. Be careful. |
| `verbose` | bool, follows project setting, but can be overwrite. *TRUE* will give you more information |
| `colorful` | bool, whether the message will be colorful? |

## Value

remove the tab file and register info in *tabs.csv*

## Examples

```
spsInit(change_wd = FALSE, overwrite = TRUE)
newTabData(
    tab_id = "data_new_remove",
    tab_displayname = "my first data tab",
    prepro_methods = list(makePrepro(label = "do nothing",
                                     plot_options = "plot_new")),
    app_path = glue::glue("SPS_{format(Sys.time(), '%Y%m%d')}")
)
removeSpsTab("data_new_remove", force = TRUE,
             app_path = glue::glue("SPS_{format(Sys.time(), '%Y%m%d')}"))
```

renderDesc *Render some collapsible markdown text*

## Description

write some text in markdown format and it will help you render to a collapsible markdown section on Shiny UI

## Usage

```
renderDesc(id, desc)
```

## Arguments

| | |
|---|---|
| id | HTML ID |
| desc | one string in markdown format |

## Value

HTML elements

## Examples

```
if(interactive()){
    library(shiny)
    desc <-
        "
    # Some desc
    - xxxx
    - bbbb

    `Some other things`
    > other markdown things
    1. aaa
    2. bbb
    3. ccc
    "
    ui <- fluidPage(
        useSps(),
        renderDesc(id = "desc", desc)
    )

    server <- function(input, output, session) {

    }

    shinyApp(ui, server)
}
```

shinyCatch *Shiny exception handling*

## Description

Exception in Shiny apps can crash the app. Most time we don't want the app to crash but just stop this block, inform users and continue with other code blocks. This function is designed to handle these issues.

## Usage

```
shinyCatch(
  expr,
  position = "bottom-right",
  blocking_level = "none",
  shiny = TRUE
)
```

## Arguments

| | |
|---|---|
| expr | expression |
| position | client side message bar position, one of: c("top-right", "top-center", "top-left","top-full-width", "bottom-right", "bottom-center", "bottom-left","bottom-full-width"). |
| blocking_level | what level you want to block the execution, one of "error", "warning", "message" |
| shiny | bool, It is also possible to use without a shiny session. Only shows on console log, works very similar as `tryCatch()` and can block at multiple levels |

## Details

Show error, warning, message by a toast bar on client end and also log the text on backend console. It will return original value if not blocking at "warning" "message" level, and return `NULL` at error level. If blocks at `error`, function will be stopped and other code in the same reactive context will be blocked. If blocks at `warning` level, warning and error will be blocked; `message` level blocks all 3 levels. The blocking works similar to shiny's `shiny::req()` and `shiny::validate()`. If anything inside fails, it will block the rest of the code in your reactive expression domain.

Messages will be displayed for 3s, 5s for warnings and errors will never go away on UI unless users' mouse hover on the bar or manually close it.

## Value

see description

## Examples

```
if(interactive()){
    ui <- fluidPage(
        useSps(),
        actionButton("btn1","error and blocking"),
        actionButton("btn2","error no blocking"),
        actionButton("btn3","warning but still returns value"),
        actionButton("btn4","warning but blocking returns"),
```

```
        actionButton("btn5","message"),

        textOutput("text")
    )
    server <- function(input, output, session) {
        fn_warning <- function() {
            warning("this is a warning!")
            return(1)
        }
        observeEvent(input$btn1, {
            shinyCatch(stop("error with blocking"), blocking_level = "error")
            print("You shouldn't see me")
        })
        observeEvent(input$btn2, {
            shinyCatch(stop("error without blocking"))
            print("I am not blocked by error")
        })
        observeEvent(input$btn3, {
            return_value <- shinyCatch(fn_warning())
            print(return_value)
        })
        observeEvent(input$btn4, {
            return_value <- shinyCatch(fn_warning(), blocking_level = "warning")
            print(return_value)
            print("other things")
        })
        observeEvent(input$btn5, {
            shinyCatch(message("message"))
        })
    }
    shinyApp(ui, server)
}
#outside shiny examples
shinyCatch(message("this message"), shiny = FALSE)
try({shinyCatch(stop("this error"), shiny = FALSE); "no block"}, silent = TRUE)
try({shinyCatch(stop("this error"), shiny = FALSE, blocking_level = "error")
    "blocked"}, silent = TRUE)
```

---

shinyCheckPkg *Shiny package checker*

---

## Description

check package name space for some required packages and pop up warnings in shiny telling users how to install the missing packages.

This is useful when some of packages are required by a shiny app. Before running into that part of code, using this function to check the required pakcage and pop up warnings will prevent app to crash.

## Usage

```
shinyCheckPkg(
  session,
  cran_pkg = NULL,
```

```
  bioc_pkg = NULL,
  github = NULL,
  quietly = FALSE
)
```

## Arguments

| | |
|---|---|
| session | shiny session |
| cran_pkg | vector of package names |
| bioc_pkg | vector of package names |
| github | vector of github packages, github package must use the format of "github user name/ repository name", eg. c("user1/pkg1", "user2/pkg2") |
| quietly | bool, should warning messages be suppressed? |

## Value

TRUE if pass, sweet alert massage and FALSE if fail

## Examples

```
if(interactive()){
    shinyApp(ui = shinyUI(
        fluidPage(actionButton("haha", "haha"))
    ), server = function(input, output, session){
        observeEvent(input$haha,
            shinyCheckPkg(session, cran_pkg = c("pkg1", "pkg2"),
                          bioc_pkg = "haha", github = "sdasdd/asdsad"))
    })
}
```

---

# sps *SystemPipeShiny app main function*

---

## Description

SystemPipeShiny app main function

## Usage

```
sps(vstabs = "", plugin = "", server_expr = NULL, app_path = getwd())
```

## Arguments

| | |
|---|---|
| vstabs | custom visualization tab IDs that you want to display, in a character vector. Use `spsTabInfo()` to see what tab IDs you can load |
| plugin | If you have loaded some SPS plugins by `spsAddPlugin()`, you can specify here as a character vector, and it will load all tabs that belong to that plugin to SPS. If you only want certain tabs from a plugin, specify in `vstabs` argument. |
| server_expr | additional top level sever expression you want to run. This will run after the default server expressions. It means you can have access to internal server expression objects, like the `shiny::reactiveValues()` object `shared`. You can also overwrite other values. Read "shared object" in vignette. |
| app_path | SPS project path |

## Details

You must set the project root as working directory for this function to find required files.

## Value

a list contains the UI and server

## Examples

```
if(interactive()){
    spsInit()
    sps_app <- sps(
        vstabs = "",
        server_expr = {
            msg("Hello World", "GREETING", "green")
        }
    )
}
```

---

`spsAddPlugin` *SPS plugin operations*

---

## Description

`spsAddPlugin()` adds an existing SPS plugin to a SPS project, `spsRemovePlugin()` is to remove a loaded plugin, and `spsNewPlugin()` is for developers to create a minimum plugin structure and required files.

## Usage

```
spsAddPlugin(
  plugin = "",
  app_path = getwd(),
  verbose = FALSE,
  third_party = FALSE,
  overwrite = 0,
  colorful = TRUE
)

spsRemovePlugin(
  plugin = "",
  app_path = getwd(),
  force = FALSE,
  verbose = FALSE,
  colorful = TRUE
)

spsNewPlugin(path, readme = TRUE, verbose = FALSE, colorful = TRUE)
```

## Arguments

| | |
|---|---|
| plugin | character, a plugin name. It can also be a path in `spsAddPlugin()` function if `third_party = TRUE`. |
| app_path | the SPS project you want to load plugin to |
| verbose | bool, show more information? |
| third_party | bool, is this an official plugin? |
| overwrite | one of 0, 1 or 2, if there are file conflicts, how to handle conflict files, see details. |
| colorful | bool, colorful message? |
| force | bool, if plugin files found, confirm before remove? |
| path | character string, path of where you want to create the plugin directory, can be a non-existing location but make sure you have write permission. |
| readme | bool, created *README.md* file? |

## Details

### General:

- You can just use `spsAddPlugin()` without any argument to see what are the plugin options.
- `plugin` should be a single name when `third_party = FALSE`, and can be a path to a custom plugin root when `third_party = TRUE`.
- Make sure there is a 'config/tabs.csv' file in your SPS project when you load the plugin.
- If there is any file conflicts when loading plugins, please see the app structure tree and refer to the legend below the tree to help you resolve conflicts. Overwriting current files is not recommended. Rename conflict files and compare them after loading the plugin will be better.
- When a plugin is removed, only tab files from that plugin are removed and entries on *config/tabs.csv* are removed. Other files come from the plugin will not be removed.
- After adding the plugin to a SPS project, you need to load it on app start, sps(..., plugin = "PLUGIN_NAME").

### overwrite mode:

- 0, if there is any conflict, abort
- 1, overwrite all overlapping files
- 2, ignore conflict files, only copy new files

### Building a plugin:

- When adding new tabs to a plugin, it will be better to set working directory to *PLUGIN_ROOT/inst/app*. Tab files should go into *PLUGIN_ROOT/inst/app/R*.
- Any additional files in *PLUGIN_ROOT/inst/app* except the *config/tabs.csv* will also be copied to users' project.
- Tab files will be checked and two functions in the tab file are expected: `tabIDUI` and `tabIDServer`. Other files will not be checked for content.
- For a plugin to work, *PLUGIN_ROOT/inst/app/config/tabs.csv* is required, and tab files listed in this *tabs.csv* are also required to be put inside *PLUGIN_ROOT/inst/app/R*. *PLUGIN_ROOT/inst/app/welcome.tx* is optional. If this file exists, content will be `cat` to console when plugin is loaded.

## Value

No return

## Examples

```
# see what official plugins you can install:
spsAddPlugin()
# create a project
spsInit(project_name = "testProject",
        change_wd = FALSE,
        open_files = TRUE,
        overwrite = TRUE)
# create a new plugin
spsNewPlugin(path = "testPlugin")
# add some tabs to the plugin
# tabs
plugin_path <- file.path("testPlugin", "inst", "app")
newTabData("data_a",
           app_path = plugin_path,
           plugin = "testPlugin",
           reformat = FALSE,
           open_file = FALSE)
newTabPlot("plot_a",
           app_path = plugin_path,
           plot_data = list(makePlotData(receive_datatab_ids = 'data_a',
                                         app_path = plugin_path)),
           plugin = "testPlugin",
           reformat = FALSE,
           open_file = FALSE)
# load the plugin, the plugin is not published, so `third_party = TRUE`
spsAddPlugin(plugin = "testPlugin",
             app_path = "testProject",
             third_party = TRUE,
             overwrite = 1)
# check if the plugin is added
# You should see `tab_vs_data_a.R` and `tab_vs_plot_a.R`
list.files(file.path("testProject", "R"))
# check if tab files are registered
# You should see the two new records
tail(vroom::vroom(file.path("testProject", "config", "tabs.csv"),
             comment = "#"))
# now remove the plugin
# now remove the plugin
# Windows connection close is delayed, may cause problems, uncomment to run
# next line before remove a plugin and try again
# closeAllConnections(); quiet(gc())
spsRemovePlugin(plugin = "testPlugin", app_path = "testProject", force = TRUE)
# let's check these files again:
list.files(file.path("testProject", "R"))
tail(vroom::vroom(file.path("testProject", "config", "tabs.csv"),
             comment = "#"))
```

---

spsDb *SPS database functions*

---

## Description

Initiate this container at global level. Methods in this class can help admin to manage general information of SPS. For now it only stores some meta data and the encryption key pairs. You can

use this database to store other useful things, like user password hash, IP, browsing info ...

A SQLite database by default is created inside `config` directory. If not, you can use `createDb` method to create one. On initiation, this class checks if the default db is there and gives warnings if not.

One instance of this class is created by the spsEncryption super class in *global.R*, normal users don't need to change anything.

## Methods

### Public methods:

- `spsDb$new()`
- `spsDb$createDb()`
- `spsDb$queryValue()`
- `spsDb$queryValueDp()`
- `spsDb$queryUpdate()`
- `spsDb$queryDel()`
- `spsDb$queryInsert()`
- `spsDb$clone()`

**Method** `new()`**:** initialize a new class object

*Usage:*

```
spsDb$new()
```

**Method** `createDb()`**:** Create a SPS database

*Usage:*

```
spsDb$createDb(db_name = "config/sps.db")
```

*Arguments:*

`db_name` database path, you need to manually create parent directory if not exists

**Method** `queryValue()`**:** Query database

*Usage:*

```
spsDb$queryValue(table, SELECT = "*", WHERE = "1", db_name = "config/sps.db")
```

*Arguments:*

`table` table name

`SELECT` SQL select grammar

`WHERE` SQL select where

`db_name` database path

*Returns:* query result, usually a dataframe

**Method** `queryValueDp()`**:** Query database with dplyr grammar

Only supports simple selections, like comparison, %in%, `between()`, `is.na()`, etc. Advanced selections like wildcard, using outside dplyr functions like [stringr::str_detect()], [base::grepl()] are not supported.

*Usage:*

```
spsDb$queryValueDp(
  table,
  dp_expr = "select(., everything())",
  db_name = "config/sps.db"
)
```

*Arguments:*

`table` table name

`dp_expr` dplyr chained expression, must use '.' in first component of the chain expression

`db_name` database path

*Returns:* query result, usually a tibble

**Method** `queryUpdate()`**:** update(modify) the value in db

*Usage:*

```
spsDb$queryUpdate(table, value, col, WHERE = "1", db_name = "config/sps.db")
```

*Arguments:*

`table` table name

`value` new value

`col` which column

`WHERE` SQL where statement, conditions to select rows

`db_name` database path

**Method** `queryDel()`**:** delete value in db

*Usage:*

```
spsDb$queryDel(table, WHERE = "1", db_name = "config/sps.db")
```

*Arguments:*

`table` table name

`WHERE` SQL where statement, conditions to select rows

`db_name` database path

**Method** `queryInsert()`**:** Insert value to db

*Usage:*

```
spsDb$queryInsert(table, value, db_name = "config/sps.db")
```

*Arguments:*

`table` table name

`value` new values for the entire row

`db_name` database path

`WHERE` SQL where statement, conditions to select rows

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
spsDb$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
dir.create("config", showWarnings = FALSE)
mydb <- spsDb$new()
mydb$createDb()
mydb$queryValue("sps_meta")
mydb$queryInsert("sps_meta", value = "'new1', '1'")
mydb$queryValue("sps_meta")
mydb$queryInsert("sps_meta", value = c("'new2'", "'2'"))
```

```
mydb$queryValue("sps_meta")
mydb$queryUpdate("sps_meta", value = '234',
                 col = "value", WHERE = "info = 'new1'")
mydb$queryValue("sps_meta")
## Not run:
    library(dplyr)
    mydb$queryValueDp(
        "sps_meta",
        dp_expr="filter(., info %in% c('new1', 'new2') %>% select(2)")

## End(Not run)
mydb$queryDel("sps_meta", WHERE = "value = '234'")
```

---

`spsEncryption` *SPS encryption functions*

---

## Description

Methods in this class can help admin to encrypt files been output from sps. For now it is only used to encypt and decrypt snapshots. This class requires the SPS database. This class inherits all functions from the spsDb class, so there is no need to initiate the `spsDb` container.

This class is required to run a SPS app. This class needs to be initialized global level. This has already been written in *global.R* for you.

## Super class

`systemPipeShiny::spsDb` -> `spsencrypt`

## Methods

### Public methods:

- `spsEncryption$new()`
- `spsEncryption$keyChange()`
- `spsEncryption$keyGet()`
- `spsEncryption$encrypt()`
- `spsEncryption$decrypt()`
- `spsEncryption$clone()`

**Method** `new()`**:** initialize a new class container

*Usage:*

```
spsEncryption$new()
```

**Method** `keyChange()`**:** Change encryption key of a SPS project

*Usage:*

```
spsEncryption$keyChange(db_name = "config/sps.db")
```

*Arguments:*

`db_name` database path

**Method** `keyGet()`**:** Get encryption key from db of a SPS project

*Usage:*

```
spsEncryption$keyGet(db_name = "config/sps.db")
```

*Arguments:*

`db_name` database path

**Method** `encrypt()`**:** Encrypt raw data or a file with key from a SPS project

*Usage:*

```
spsEncryption$encrypt(
  data,
  out_path = NULL,
  overwrite = FALSE,
  db_name = "config/sps.db"
)
```

*Arguments:*

`data` raw vector or a file path

`out_path` if provided, encrypted data will be write to a file

`overwrite` if `out_path` file exists, overwrite?

`db_name` database path

**Method** `decrypt()`**:** Decrypt raw data or a file with key from a SPS project

*Usage:*

```
spsEncryption$decrypt(
  data,
  out_path = NULL,
  overwrite = FALSE,
  db_name = "config/sps.db"
)
```

*Arguments:*

`data` raw vector or a file path

`out_path` if provided, encrypted data will be write to a file

`overwrite` if `out_path` file exists, overwrite?

`db_name` database path

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
spsEncryption$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
dir.create("config", showWarnings = FALSE)
spsOption('verbose', TRUE)
my_ecpt <- spsEncryption$new()
my_ecpt$createDb()
my_ecpt$keyChange()
# imagine a file has one line "test"
writeLines(text = "test", con = "test.txt")
# encrypt the file
```

```
my_ecpt$encrypt("test.txt", "test.bin", overwrite = TRUE)
# decrypt the file
my_ecpt$decrypt("test.bin", "test_decpt.txt", overwrite = TRUE)
# check the decrypted file content
readLines('test_decpt.txt')
```

---

spsHr *hr line with color #3b8dbc38*

---

## Description

hr line with color #3b8dbc38

## Usage

```
spsHr()
```

## Value

HTML elements

## Examples

```
spsHr()
```

---

spsInit *Create a SystemPipeShiny project*

---

## Description

To run a SPS app, you need to first create a SPS project, a directory contains the required files.

## Usage

```
spsInit(
  app_path = getwd(),
  project_name = glue::glue("SPS_{format(Sys.time(), '%Y%m%d')}"),
  database_name = "sps.db",
  overwrite = FALSE,
  change_wd = TRUE,
  verbose = FALSE,
  open_files = TRUE,
  colorful = TRUE
)
```

### Arguments

| | |
|---|---|
| app_path | path, a directory where do you want to create this project, must exist. |
| project_name | Your project name, default is SPS_ + time |
| database_name | project database name, recommend to use the default name: "sps.db". It is used to store app meta information, see spsDb() |
| overwrite | bool, overwrite the app_path if there is a folder that has the same name as project_name? |
| change_wd | bool, when creation is done, change working directory into the project? |
| verbose | bool, do you want additional message? |
| open_files | bool, If change_wd == TRUE and you are also in Rstudio, it will open up *global.R* for you |
| colorful | bool, should message from this function be colorful? |

### Details

Make sure you have write permission to app_path

### Value

creates the project folder

### Examples

```
if(interactive()){
    spsInit(change_wd = FALSE)
}
```

---

## spsOption *Get or set SPS options*

---

### Description

Get or set SPS options

### Usage

```
spsOption(opt, value = NULL, empty_is_false = TRUE)
```

### Arguments

| | |
|---|---|
| opt | string, length 1, what option you want to get or set |
| value | if this is not NULL, this function will set the option you choose to this value |
| empty_is_false | bool, when trying to get an option value, if the option is NULL, NA, "" or length is 0, return FALSE? |

### Value

return the option value if value exists; return FALSE if the value is empty, like NULL, NA, ""; return NULL if empty_is_false = FALSE; see emptyIsFalse

If value != NULL will set the option to this new value, no returns.

## See Also

`viewSpsDefaults()` for options you can view or set

## Examples

```
spsOption("test1", 1)
spsOption("test1")
spsOption("test2")
spsOption("test2", empty_is_false = FALSE)
```

---

`spsTabInfo` *View SPS project 'config/tabs.csv' information*

---

## Description

View SPS project 'config/tabs.csv' information

## Usage

```
spsTabInfo(return_type = "print", n_print = 40, app_path = getwd())
```

## Arguments

| | |
|---|---|
| `return_type` | one of 'print', 'data', 'colnames', or a specified column name |
| `n_print` | how many lines of tab info you want to print out |
| `app_path` | SPS project root |

## Details

- 'print' will print out the entire *tabs.csv*, you can specify `n_print` for how many lines you want to print;
- 'data' will return the tab info tibble
- 'colnames' will return all column names of tab info file
- A column name will extract the specified column out and return as a vector

## Value

return depends on `return_type`

## Examples

```
spsInit(project_name = "SPS_tabinfo", overwrite = TRUE,
        change_wd = FALSE, open_files = FALSE)
# all lines
spsTabInfo("print", app_path = "SPS_tabinfo")
# 5 lines
spsTabInfo("print", app_path = "SPS_tabinfo", n_print = 5L)
spsTabInfo("data", app_path = "SPS_tabinfo")
spsTabInfo("colnames", app_path = "SPS_tabinfo")
spsTabInfo("tab_id", app_path = "SPS_tabinfo")
```

spsValidate *Validate expressions*

## Description

this function is usually used on server side to validate input dataframe or some expression

## Usage

```
spsValidate(
  expr,
  vd_name = "validation",
  pass_msg = glue("{vd_name} passed"),
  fail_msg = glue("{vd_name} failed"),
  shiny = TRUE,
  verbose = spsOption("verbose")
)
```

## Arguments

| | |
|---|---|
| expr | the expression to validate data or other things. It should return TRUE if pass or use stop("your message") if fail. Other types of return are acceptable but not recommended. As long as it is not empty or FALSE by the emptyIsFalse() function, it will return TRUE |
| vd_name | validate title |
| pass_msg | string, if pass, what message do you want to show |
| fail_msg | string, optional, if your expression does not contain the use of stop() for failure and only returns FALSE or other empty values, this message will show and generate shiny reactive stop |
| shiny | you can use this function outside shiny, see shinyCatch() for more |
| verbose | bool, show pass message? Default follows project verbose setting |

## Value

If expression returns empty or FALSE make it shiny reactive stop and no final return, else TRUE.

## See Also

shinyCatch, emptyIsFalse

## Examples

```
spsOption("verbose", TRUE)
if(interactive()){
    ui <- fluidPage(
        useSps(),
        actionButton("vd1", "validate1"),
        actionButton("vd2", "validate2")
    )
    server <- function(input, output, session) {
        mydata <- datasets::iris
```

```
        observeEvent(input$vd1, {
            spsValidate({
                is.data.frame(mydata)
            }, vd_name = "Is df")
            print("continue other things")
        })
        observeEvent(input$vd2, {
            spsValidate({
                nrow(mydata) > 200
            }, vd_name = "more than 200 rows")
            print("other things blocked")
        })
    }
    shinyApp(ui, server)
}
# outside shiny example
mydata2 <- list(a = 1, b = 2)
spsValidate({(mydata2)}, "Not empty", shiny = FALSE)
try(spsValidate(is.data.frame(mydata2),
                "is dataframe?",
                shiny = FALSE),
    silent = TRUE)
```

---

`tabTitle` *h2 title with bootstrap info color*

---

## Description

h2 title with bootstrap info color

## Usage

```
tabTitle(title, ...)
```

## Arguments

| | |
|---|---|
| `title` | title text |
| `...` | other attributes and children to this element |

## Value

a h2 level heading with bootstrap4 "info" color(bt4 color not the default bt3 info color)

## Examples

```
tabTitle("This title")
```

## textInputGroup *Text input with a button right next to*

### Description

Text input with a button right next to

### Usage

```
textInputGroup(textId, btnId, title = "", label = "", icon = "paper-plane")
```

### Arguments

| | |
|---|---|
| textId | text box id |
| btnId | action button id |
| title | title of this group |
| label | button text |
| icon | button icon |

### Value

a row element

### Examples

```
if(interactive()){

    ui <- fluidPage(
        useSps(),
        textInputGroup("id1", "id2")
    )

    server <- function(input, output, session) {

    }

    shinyApp(ui, server)
}
```

## uiExamples *Example UI elements for plotting*

### Description

return some example UI elements can be toggled on plotting. This functions is only used as a temp solution for the example tab to demonstrate what UI components you can use. Will be removed in later releases as we have a better tab organizations.

### Usage

```
uiExamples(ns)
```

### Arguments

| | |
|---|---|
| ns | namespace function |

### Value

some UI

### Examples

```
if(interactive()){
    ui <- fluidPage(useSps(), uiExamples(NS("example")))
    server <- function(input, output, session) {}
    shinyApp(ui, server)
}
```

---

## useSps *Use SystemPipeShiny javascripts and css style*

---

### Description

call it in your head section of your shiny UI. Adding this to the Shiny app UI is required for most SPS shiny widgets. This is required for using SPS widgets outside of SPS framework. No need to load this function again if you are working within SPS framework.

### Usage

```
useSps()
```

### Value

HTML head

### Examples

```
useSps()
```

`viewSpsDefaults` *Print SPS default options*

## Description

Make sure you have created the app directory and it has *config/config.yaml* file

## Usage

```
viewSpsDefaults(app_path = getwd())
```

## Arguments

| | |
|---|---|
| `app_path` | path, where is the app directory |

## Value

cat to console the default options

## Examples

```
if(interactive()){
    spsInit(open_files = FALSE)
    viewSpsDefaults()
}
```

# Index