# Package 'systemPipeShiny'

October 24, 2025

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

**Version** 1.19.1

**Date** 2025-05-19

**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 (>= 1.6.0), spsUtil (>= 0.2.2), spsComps (>= 0.3.3), drawer (>= 0.2)

**Imports** DT, assertthat, bsplus, crayon, dplyr, ggplot2, htmltools, glue, magrittr, methods, plotly, rlang, rstudioapi, shinyAce, shinyFiles, shinyWidgets, shinydashboard, shinydashboardPlus (>= 2.0.0), shinyjqui, shinyjs, shinytoastr, stringr, stats, styler, tibble, utils, vroom (>= 1.3.1), yaml, R6, RSQLite, openssl

**Suggests** testthat, BiocStyle, knitr, rmarkdown, systemPipeR (>= 2.12.0), systemPipeRdata (>= 2.10.0), rhandsontable, zip, callr, pushbar, fs, readr, R.utils, DESeq2, SummarizedExperiment, glmpca, pheatmap, grid, ape, Rtsne, UpSetR, tidyr, esquisse (>= 1.1.0), cicerone

**VignetteBuilder** knitr

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

**License** GPL (>= 3)

**Encoding** UTF-8

**LazyData** true

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

**URL** https://systempipe.org/sps, https://github.com/systemPipeR/systemPipeShiny

**RoxygenNote** 7.3.2

**Roxygen** list(markdown = TRUE)

**Config/testthat/edition** 3

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

**git_branch** devel

**git_last_commit** c93c6f8

**git_last_commit_date** 2025-05-20

**Repository** Bioconductor 3.22

**Date/Publication** 2025-10-23

**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>

# Contents

## canvasBtn *Screenshot a plot or UI to SPS Canvas or download as an image*

### Description

A upper level function of drawer::toCanvasBtn. You should only use it under SPS projects. For you own apps, still use the drawer::toCanvasBtn.

### Usage

```
canvasBtn(dom, id = "", isID = TRUE, class = "text-center", placement = "top")
```

### Arguments

| | |
|---|---|
| dom | a HTML DOM selector, mostly common is to select the element by ID:<br>e.g. a plot with ID "plot1", to select, use `dom = "plot1"` to select the plot if `isID = TRUE`. If `isID = FALSE`, use `dom = "#plot1"`<br>Other complex selector is supported. First turn `isID = FALSE`, then try things like `dom = ".btn i"` selects an icon inside an element with "btn" class. If more than one element is matched, only the first one will be screenshoted. |
| id | ID of this button, optional. |
| isID | bool, if the `dom` argument is selected by ID or other selector |
| class | string, length 1, other html class add to the button wrapper |
| placement | where should the tiptool place, top, bottom, left, right. |

### Value

a button group with several options

### Examples

```
canvasBtn("#mydiv")
```

## dynamicFile *Dynamically generate Shiny file selection component based on option*

### Description

Depending on the "mode" in SPS options, this function renders a similar UI components but behaves differently on server.

1. `local` mode will not copy file, directly use a path pointer.
2. `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,
  buttonType = "primary",
  placeholder = "No file selected",
  mode = spsOption("mode")
)

dynamicFileServer(
  input,
  session,
  id,
  mode = spsOption("mode"),
  roots = c(root = "default")
)
```

## Arguments

| | |
|---|---|
| `id` | element ID, Use `ns()` to wrap the id if you are using within a shiny module, but DO NOT use `ns()` to wrap the id on server side |
| `title` | element title |
| `label` | upload button label |
| `icon` | button icon, an object create by shiny::icon |
| `style` | additional button style, only works for `local` mode |
| `multiple` | bool, are multiple files allowed? |
| `buttonType` | string, Bootstrap button markup (color). Default in SPS is 'primary', other valid values include 'info', 'success', 'default', 'warning', 'danger'. |
| `placeholder` | string, text to display before the file is uploaded |
| `mode` | string, one of "local" or "server" |
| `input` | shiny server input |
| `session` | shiny server session |
| `roots` | a named character vector, paths where users can reach on the server, so only required for "server" mode, default is current directory + all system volumes. You can lock users to a specific path, so they are not allowed to browse parent folders. like only current directory: `c(current=getwd())`; a temp folder: `c(current=tempdir())`; unlimited: `c(shinyFiles::getVolumes()())` |

## Details

To setup the option:

The `local` mode uses functions from shinyFiles so it will reach file system on the **server end**. Although the latest shinyFiles limits users to only specified server end location (folder), there is still some **risk**. That's why it is named "local", you are encouraged to run the app on your local computer. The advantage of "local" is: for some very large files, it does not upload and store in the

temp. Rather, it directly parses the path on the local file system and return the path immediately. It means the file has to exist on the file system that serves the Shiny app. If you deploy the app on places like shinyapps.io, users can only choose files from server.

On the other hand, `server` mode uses original but enhanced shiny default upload component. Users can upload files from local to server. So users do not have access to server end file system if you deploy it online. However, the limitations are:

1. not ideal for large files, default limit is 30MB, and there is no break-point upload.
2. If you are running the app on your own computer, local end and server end is the same, which is your computer. Using `server` mode will make a copy of your existing file to temp location and this is a waste of time and storage.

To set up options:

1. Under SPS framework, edit options in `global.R`.
2. Outside SPS framework with your own Shiny app, use `spsUtil::spsOption()` function, like `spsUtil::spsOption("mode", "server")` or `spsUtil::spsOption("mode", "local")` to set up mode.

If you are not sure what mode you are on, use `spsUtil::spsOption('mode')` to check.

## Value

a Shiny upload component on UI

For the server end it returns a **reactive** object which is a dataframe, need to extract the value inside reactive expression, observe, or inside `isolate`. See examples

## Examples

```
# Simple example
if(interactive()){
    spsOption("mode", value = "server") # Change the value to 'local' to see difference
    ui <- fluidPage(
        dynamicFile(id = "server_file", label = "server"),
        verbatimTextOutput("server_out")
    )

    server <- function(input,output,session){
        file_server <- dynamicFileServer(input,session, id = "server_file")
        output$server_out <- renderPrint({
            file_server()  # remember to use `()` for reactive value
        })
    }
    shinyApp(ui = ui, server = server)
}
# To demostrate different modes in the same app, we can set options before the function.
# This is NOT recommended, you should stick with only one mode for the entire app.
if(interactive()){
    spsOption("mode", "local")
    local_ui <- dynamicFile("local_file", "local")
    spsOption("mode", "server")
    server_ui <- dynamicFile("server_file", "server")
    ui <- fluidPage(
        column(
            6,
```

```
            local_ui,
            verbatimTextOutput("local_out")
        ),
        column(
            6,
            server_ui,
            verbatimTextOutput("server_out")
        )
    )

    server <- function(input,output,session){
        spsOption("mode", "local")
        file_local <- dynamicFileServer(input,session, id = "local_file")
        output$local_out <- renderPrint({
            file_local() # remember to use `()` for reactive value
        })
        spsOption("mode", "server")
        file_server <- dynamicFileServer(input,session, id = "server_file")
        output$server_out <- renderPrint({
            file_server()
        })
    }
    shinyApp(ui = ui, server = 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 spsComps::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)
}
```

---

# genHrefTable *Generate a table that lists tabs by rows*

---

## Description

A fast way in SPS to generate a table that lists some 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 spsComps::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)
}
```

---

# loadDF *Load tabular files as tibbles to server*

---

## Description

load a file to server end. It's designed to be used with a input file source switch button. 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.

If no user data is uploaded, it will return the example dataset that is prepared by the developer. If the developer does not provide the dataset either, it will return a 8-row empty tibble.

## 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, one of '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 to parse the datafile, see help file of vroom::vroom |
| delim | delimiter characters to parse the data file, see help file of vroom::vroom |
| col_types | columns specifications, see help file of vroom::vroom |
| ... | other params for vroom, see help file of vroom::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 loading file errors to crash the Shiny app, so any kind of file upload will not crash the app. To show message on UI, `spsDepend("toastr")` must be used in Shiny UI function, see examples.

## Value

returns a tibble and NULL if parsing fails

## Examples

```
if(interactive()){
  # change value to 'local' to see the difference
  spsOption("mode", value = "server")
  ui <- fluidPage(
    spsDepend("toastr"),
    radioButtons(
      "data_source", "Choose your data file source:",
      c("Upload" = "upload", "Example" = "eg"),
      selected = "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)
}
```

---

`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)
spsNewTab("vs_new", app_path = glue::glue("SPS_{format(Sys.time(), '%Y%m%d')}"))
removeSpsTab("vs_new", force = TRUE,
             app_path = glue::glue("SPS_{format(Sys.time(), '%Y%m%d')}"))
```

---

sps *SystemPipeShiny app main function*

---

## Description

SystemPipeShiny app main function

## Usage

```
sps(
  tabs = "",
  server_expr = NULL,
  login_message = shiny::h3("User login"),
  app_path = getwd()
)
```

## Arguments

| | |
|---|---|
| tabs | custom visualization tab IDs that you want to display, in a character vector. Use `spsTabInfo()` to see what tab IDs you can load |
| 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 manual. |
| login_message | a shiny tag that will be displayed on the top of login panel, default is a H3 title with text "User login", `shiny::h3("User login")`. If you need more information, you can do something like `div(h3("Login"), p("Some more message"))`. |
| app_path | SPS project path |

## Details

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

**About this function:**

Usually you call this function inside the *global.R* file when SPS initialization is done. This function does not contain too many options. Most choices are controlled by SPS options which are also listed in *global.R* (some lines before calling this function in that file).

## Value

a list contains the UI and server

## Examples

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

---

`spsAccount` *SPS account management functions*

---

## Description

Initiate this container at global level. Methods in this class can help admins to manage accounts in a SPS project.

It uses a SQLite database, by default is created inside `config` directory on SPS initialization.

You can use it to add/remove users, change user roles, change password, match/verify account, password, role.

A default user account "user", with password "user", and a default admin account "admin" with password "admin" are create for you.

For app deployment, PLEASE create your own accounts and DELETE the default ones.

## Super classes

`systemPipeShiny::spsDb` -> `systemPipeShiny::spsEncryption` -> `spsaccount`

## Methods

### Public methods:

- `spsAccount$new()`
- `spsAccount$accList()`
- `spsAccount$accAdd()`
- `spsAccount$accRemove()`
- `spsAccount$accPassChange()`
- `spsAccount$accRoleChange()`
- `spsAccount$accMatch()`
- `spsAccount$clone()`

**Method** `new()`**:** initialize a new SPS account container

*Usage:*

```
spsAccount$new()
```

**Method** `accList()`**:** list all accounts of the app. Returns a dataframe

*Usage:*

```
spsAccount$accList(include_pass = FALSE, db_name = "config/sps.db")
```

*Arguments:*

`include_pass` bool, include password hash column?

`db_name` SPS database path

**Method** `accAdd()`**:** add an account to use the app

*Usage:*

```
spsAccount$accAdd(acc_name, acc_pass, role = "user", db_name = "config/sps.db")
```

*Arguments:*

`acc_name` string, account name

`acc_pass` string, account password

`role` string, what kind role is this user, one of "user", "admin"

`db_name` SPS database path

**Method** `accRemove()`**:** remove an account

*Usage:*

```
spsAccount$accRemove(acc_name, db_name = "config/sps.db")
```

*Arguments:*

`acc_name` string, account name

`db_name` SPS database path

**Method** `accPassChange()`**:** change password of an account

*Usage:*

```
spsAccount$accPassChange(acc_name, acc_pass, db_name = "config/sps.db")
```

*Arguments:*

`acc_name` string, account name

`acc_pass` string, account new password

`db_name` SPS database path

**Method** `accRoleChange()`**:** change the role of an account

*Usage:*

```
spsAccount$accRoleChange(acc_name, role, db_name = "config/sps.db")
```

*Arguments:*

`acc_name` string, account name

`role` string, one of "user" or "admin"

`db_name` SPS database path

**Method** `accMatch()`**:** Try to see if the account name exists and has the right password and role type, useful for login authentification.

*Usage:*

```
spsAccount$accMatch(
  acc_name,
  acc_pass,
  role = "user",
  match_role = FALSE,
  db_name = "config/sps.db"
)
```

*Arguments:*

`acc_name` string, account name
`acc_pass` string, account new password
`role` string, one of "user" or "admin"
`match_role` bool, also verify the account role type?
`db_name` SPS database path

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

*Usage:*

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

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
dir.create("config", showWarnings = FALSE)
spsOption("verbose", TRUE)
spsOption("use_crayon", TRUE)
# create a new container
db <- spsAccount$new()
db$createDb()
# list all accounts
db$accList()
# add a new user
db$accAdd('user2', '!admin12345')
# list all accounts include password hash
db$accList(include_pass = TRUE)
# change password of an account
db$accPassChange("user2", "$aaaaaaa")
# check if pass changed
db$accList(include_pass = TRUE)
# change the role of from user to admin
db$accRoleChange("user2", "admin")
# check role change
db$accList()
# remove a user
db$accRemove("user2")
# check accounts again
db$accList()
# check if username and password matches
db$accMatch(acc_name = "user", acc_pass = "user")
# wrong pass
db$accMatch("user", "user123")
# also check if the user has the right role
db$accMatch("user", "user", role = "user", match_role = TRUE)
db$accMatch("user", "user", role = "admin", match_role = TRUE)
```

---

# spsCoreTabReplace *Overwrite a default SPS tab*

---

## Description

If you want to load your custom content on any of the default tabs in a SPS project, you can overwrite the tab with your own UI and server function. First, use this function to create a template for the tab you want to replace and then fill your own content.

## Usage

```
spsCoreTabReplace(
  replace_tab,
  app_path = getwd(),
  open_file = TRUE,
  overwrite = FALSE
)
```

## Arguments

| | |
|---|---|
| replace_tab | one of "welcome", "module_main", "vs_main", "canvas", "about", for the welcome tab, module home tab, custom tab home tab, Canvas tab, about tab respectively. |
| app_path | string, where is SPS project root path |
| open_file | bool, open the newly created template if you are in Rstudio? |
| overwrite | bool, if the template exists, overwrite it with a new, empty one? |

## Value

a template file

## Examples

```
if(interactive()){
    spsInit(project_name = "default_overwrite_demo",
            change_wd = FALSE, open_files = FALSE)
    ## try to run it for the first look
    # shiny::runApp("default_overwrite_demo")
    spsCoreTabReplace("welcome", app_path = "default_overwrite_demo")
    ## edit the file and save it.
    ## run again and watch the difference on the welcome tab.
    shiny::runApp("default_overwrite_demo")
}
```

---

# 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 stores some meta data, the encryption key pairs and the account info. 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 spsAccount 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 where grammar

`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, collect all values from all columns in a vector.

`db_name` database path

**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` -> `spsEncryption`

## 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(confirm = FALSE, db_name = "config/sps.db")
```

*Arguments:*

`confirm`, bool, confirm that you understand the consequence

`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()
# Read carefully before change the key
my_ecpt$keyChange()
# confirm
my_ecpt$keyChange(confirm = TRUE)
# 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')
```

spsEzUI *Easy and simple UI and server for a SPS custom tab*

## Description

SPS custom tab simple UI and server , spsEzUI must use together with the spsEzServer function. The easiest way to use is to use spsNewTab function to create both.

## Usage

```
spsEzUI(
  desc = "",
  tab_title = "Tab Title",
  plot_title = "My Plot",
  plot_control = shiny::tagList()
)

spsEzServer(
  plot_code,
 example_data_path = system.file(package = "systemPipeShiny", "app", "data", "iris.csv"),
  other_server_code = ""
)
```

## Arguments

| | |
|---|---|
| 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
"
```

| | |
|---|---|
| tab_title | string, tab title |
| plot_title | string, plot title |
| plot_control | some Shiny components (UI) to control the plot, like plot title, x,y labels, color, font size, etc. Group all components in a shiny tagList. |
| plot_code | some R code to make the plot. |
| example_data_path | example dataset path, this dataset will be loaded on app start to display |
| other_server_code | optional, other server R code to run for this tab |

## Value

spsEzUI returns a shiny module UI function, spsEzServer returns the server function

## See Also

spsNewTab

## Examples

```
# use `spsInit()` to create an SPS project and use `spsNewTab("Your_tabID", template = "easy")`
# to create a new tab file. The specified use of these two functions is in that file.
```

---

`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` | deprecated in current version. project database name, recommend to use the default name: "sps.db". It is used to store app meta information. |
| `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`.

The database in not used in current version.

## Value

creates the project folder

## Examples

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

---

spsNewTab *Create a new SPS tab*

---

## Description

create custom tabs in newer (> 1.1) version of SPS. The old creation functions will be deprecated by next Bioconductor major release.

## Usage

```
spsNewTab(
  tab_id = "vs_mytab",
  tab_displayname = "My custom plotting tab",
  img = "",
  app_path = getwd(),
  out_folder_path = file.path(app_path, "R"),
  author = "",
  template = c("simple", "full"),
  preview = FALSE,
  reformat = FALSE,
  open_file = TRUE,
  verbose = spsOption("verbose"),
  colorful = spsOption("use_crayon")
)
```

## Arguments

| | |
|---|---|
| tab_id | character string, length 1, must be unique. 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 |
| img | realtive path, an image representation of the new plot. It can be a internet link or a local link which uses the *www* folder as the root. e.g. drop your image *plot.png* inside *www/plot_list*, then the link here is "plot_list/plot.png". You will see these images on "Custom Tabs" main page. If no provided, a warning will be given on app start and an empty image will show up on "Custom Tabs". |
| 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. If you write the file other than *R*, this file will not be automatically loaded by SPS or Shiny. You must source it manually. |
| author | character string, or a vector of strings. authors of the tab |

| | |
|---|---|
| template | one of "simple" or "full", default "simple". "simple" gives a tab file with minimum Shiny code, you can only focus on you R plotting code. "full" gives the full tab code, so you can modify everything on the tab. |
| preview | bool, *TRUE* will print the new tab code to console and will not write the file and will not register the tab |
| 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 |

## Details

- template "simple": hides the UI and server code and use spsEzUI and spsEzServer instead.
- template "full": full tab code. You need to know some Shiny development knowledge.

## Value

returns a new tab file

## Examples

```
spsInit(change_wd = FALSE, overwrite = TRUE)
spsNewTab("vs_newtab_ez", app_path = glue::glue("SPS_{format(Sys.time(), '%Y%m%d')}"))
spsNewTab("vs_newtab_full", template = "full",
app_path = glue::glue("SPS_{format(Sys.time(), '%Y%m%d')}"))
spsNewTab("vs_newtab_pre", preview = TRUE,
app_path = glue::glue("SPS_{format(Sys.time(), '%Y%m%d')}"))
```

---

# spsOptDefaults *Print SPS options*

---

## Description

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

spsOptDefaults prints out all default and other avaliable values for each option. spsOptions print all current set option values.

Note: the spsUtil::spsOption is used to get or set a **single** option value. spsOptions is used to print **all** current option values. If you need to set all values at once, use the *global.R* file under SPS project root.

## Usage

```
spsOptDefaults(app_path = getwd())

spsOptions(app_path = getwd(), show_legend = TRUE)
```

## Arguments

| | |
|---|---|
| app_path | path, where is the app directory |
| show_legend | bool, show the color legend? |

## Value

cat to console SPS option values

## Examples

```
if(interactive()){
    # start a SPS project
    spsInit(open_files = FALSE)
    viewSpsDefaults()
    # change a few options
    options(sps = list(
        mode = "server",
        warning_toast = TRUE,
        login_screen = FALSE,
        login_theme = "vhelix",
        use_crayon = TRUE
    ))
    # view current options
    spsOptions()
}
```

---

# 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")
```

# Index