Package 'shinyEventLogger'

February 22, 2019

Type Package

Title Logging Events in Shiny Apps

Version 0.1.1

Description Logging framework dedicated for complex shiny apps.
Different types of events can be logged
(value of a variable, multi-line output of a function,
result of a unit test, custom error, warning, or diagnostic message).
Each event can be logged with a list of parameters that are event-specific,
common for events within the same scope, session-specific, or app-wide.
Logging can be done simultaneously to R console,
browser JavaScript console, a file log, and a database (MongoDB).
Log data can be further analyzed with the help of process-mining techniques
from 'bupaR' package.

URL https://github.com/kalimu/shinyEventLogger#readme,
https://kalimu.github.io/project/shinyeventlogger/

BugReports https://github.com/kalimu/shinyEventLogger/issues

License MIT + file LICENSE

Encoding UTF-8

Language en-US

Imports shiny, jsonlite, mongolite, R.utils, utils, stringr, purrr,
bupaR

Suggests spelling, DiagrammeR, testthat, knitr, rmarkdown

LazyData true

RoxygenNote 6.1.1

VignetteBuilder knitr

NeedsCompilation no

Author Kamil Wais [aut, cre] (<https://orcid.org/0000-0002-4062-055X>)

Maintainer Kamil Wais <kamil.wais@gmail.com>

Repository CRAN

Date/Publication 2019-02-22 10:20:03 UTC

R topics documented:

`inspect_object` *Copying objects to global environment*

Description

Convenient wrapper for `inspect_objects`.

Usage

```
inspect_object(...)
```

Arguments

...	Named objects to be copy to the global environment. If there is only one unnamed object, its name in the global environment will be the same as the name of the object passed to ... .

inspect_objects *Copying objects to global environment*

Description

With `inspect_objects` you can copy an object to the global environment for further debugging or developing.

Usage

```
inspect_objects(...)
```

Arguments

`...`	Named objects to be copy to the global environment. If there is only one unnamed object, its name in the global environment will be the same as the name of the object passed to `....`

Examples

```
if (interactive()) {

  set_logging()
  shiny::shinyApp(
    ui = shiny::fluidPage(log_init()),
    server = function(input, output) {
      set_logging_session()
      inspect_objects(mtcars)
      inspect_objects(df1 = head(mtcars), df2 = head(iris))
    }
  )

}
```

log_event *Logging an event*

Description

`log_event` logs an event into R console, browser JavaScript console, file, or database depending on user's settings (see `set_logging`).

Usage

```
log_event(..., name = NULL, type = "EVENT", status = "FIRED",
  params = NULL, event_counter = NULL)
```

Arguments

...	Objects that are evaluated, coerced into character string, collapsed and pasted into log entry body (or header if name is NULL).
name	A character string. The name of the event.
type	A character string. A type of the event. Default is "EVENT".
status	A character string. A status of the event. Default is "FIRED".
params	A list of additional named event-specific parameters. Default is NULL.
event_counter	An integer. The number of the event. Default is NULL which will be replaced by the current value of the counter returned by the internal getter function get_event_counter.

See Also

set_logging for setting event logging, log_init for initialize JavaScript logging in shiny app, log_params for setting scope-specific event parameters, read_eventlog for reading eventlog from a file or a database.

Other logging events functions: log_message, log_output, log_started, log_test, log_value

Examples

```
if (interactive()) {
  set_logging()
  shiny::shinyApp(
    ui = shiny::fluidPage(log_init()),
    server = function(input, output) {
      set_logging_session()
      log_event("Event 1")
      log_event("Event 2 body", name = "Event 2")
      log_event("Event 3", type = "NewTYPE")
      log_event("Event 4", status = "EXECUTED")
      log_event("Event 5", event_counter = 123)
    }
  )
}
```

---

log_init *Initialize logging to JavaScript console*

Description

log_init should be put into the shiny ui to initialize JavaScript code that enables logging to JavaScript console in an Internet browser.

Usage

```
log_init()
```

Value

A tagList with script tag inside head tag.

Examples

```
if (interactive()) {
  set_logging(js_console = TRUE)
  shiny::shinyApp(
    ui = shiny::fluidPage(log_init()),
    server = function(input, output) {
      set_logging_session()
      log_event("See browser JavaScript console (CTRL + SHIFT + I)")
    }
  )
}
```

---

log_message *Logging a message*

Description

log_message, log_warning, and log_error are wrapper functions for logging events of type MESSAGE, WARNING, or ERROR. Relevant message, warning or error is raised after logging an event. Raising an error is done using stop function and it can stop the whole shiny app.

Usage

```
log_message(...)

log_warning(...)

log_error(...)
```

Arguments

...	Objects that are evaluated, coerced into character string, collapsed and pasted as event name into log entry header. The character string is also passed to the message, warning, or error raised.

Functions

- log_message: Logging a message
- log_warning: Logging a warning
- log_error: Logging an error

See Also

Other logging events functions: log_event, log_output, log_started, log_test, log_value

Examples

```
if (interactive()) {
  set_logging()
  shiny::shinyApp(
    ui = shiny::fluidPage(log_init()),
    server = function(input, output) {
      set_logging_session()
      log_message("Example of a message")
      log_warning("Example of a warning")
      log_error("Example of an error")
    }
  )
}
```

---

log_output *Logging output of a function*

Description

`log_output` logs output of a function into log entry body and uses deparsed function call as the event name.

Usage

```
log_output(..., type = "OUTPUT", status = "FIRED", params = NULL)
```

Arguments

`...`	A function call that is evaluated, coerced into character string, collapsed and pasted as multi-line text into log entry body. Deparsed function call is used as the event name in log entry header.
`type`	A character string. A type of the event. Default is `"OUTPUT"`.
`status`	A character string. A status of the event. Default is `"FIRED"`.
`params`	A list of additional named event-specific parameters. Default is `NULL`.

See Also

Other logging events functions: `log_event`, `log_message`, `log_started`, `log_test`, `log_value`

Examples

```
if (interactive()) {
  set_logging()
  shiny::shinyApp(
    ui = shiny::fluidPage(log_init()),
    server = function(input, output) {
      set_logging_session()
      log_output(NROW(mtcars))
```

```
      log_output(head(mtcars))
    }
  )

}
```

`log_params` *Logging scope-specific parameters of events*

Description

With `log_params` you can define a set of named parameters, which are common for events from the same scope (for example inside an observer). These parameters will be added to event-specific parameters and logged within the same log entry.

Usage

```
log_params(...)
```

Arguments

`...`	a set of named objects (usually of type character, numeric, or date) to be logged as event parameters.

Details

The function takes all objects passed inside `...` argument, evaluates them, and stores them in a new environment called `log_setting` which is assigned to the parent environment from which the `log_params` function was called.

See Also

Other setting up logging parameters functions: `set_logging_session`, `set_logging`

Examples

```
if (interactive()) {
  set_logging()
  shiny::shinyApp(
    ui = shiny::fluidPage(log_init()),
    server = function(input, output) {
      set_logging_session()
      observe({
        log_params("observer" = "A")
        log_event("Event A.1")
        log_event("Event A.2")
      })
      observe({
        log_params("observer" = "B")
```

```
        log_event("Event B.1")
        log_event("Event B.2")
      })
    }
  )
}
```

`log_started` *Logging the start of an event*

Description

`log_started` logs an event with status `"STARTED"`. `log_done` logs the same event with status `"DONE"`. Difference between timestamps of these two log entries can be used for timing an event. One event can have several instances with different statuses. When logging instances of the same event, event `name` or, if `name = NULL`, objects passed to `...` must be exactly the same, as they are used to create unique event id. Started events, their types and counters are registered in an environment called `log_event_register`, which enables creating and timing multiple nested events.

Usage

```
log_started(..., name = NULL, type = "EVENT", status = "STARTED",
  params = NULL)

log_done(..., name = NULL, params = NULL)
```

Arguments

`...`	Objects that are evaluated, coerced into character string, collapsed and pasted into log entry body (or header if `name` is `NULL`).
`name`	A character string. The name of the event.
`type`	A character string. A type of the event. Default for `log_started` is `"EVENT"`. The `type` logged with `log_done` is the same as the `type` of the event logged with `log_started`.
`status`	A character string. A status of the event. Default for `log_started` is `"STARTED"`. The `status` is always `"DONE"` when using `log_done`.
`params`	A list of additional named event-specific parameters. Default is `NULL`.

Functions

- `log_started`: Logging the start of an event
- `log_done`: Logging the end of an event

See Also

Other logging events functions: `log_event`, `log_message`, `log_output`, `log_test`, `log_value`

Examples

```
if (interactive()) {
  set_logging()
  shiny::shinyApp(
    ui = shiny::fluidPage(log_init()),
    server = function(input, output) {
      set_logging_session()
        log_started(as.character(Sys.time()), name = "Event 1")
        Sys.sleep(0.5)
          log_started(as.character(Sys.time()), name = "Event 2")
            log_event(as.character(Sys.time()), name = "Event 3")
            Sys.sleep(0.5)
          log_done(as.character(Sys.time()), name = "Event 2")
        log_done(as.character(Sys.time()), name = "Event 1")
    }
  )
}
```

---

log_test *Logging unit test*

Description

log_test logs a unit test which can be built in inside a shiny app. The event logged has status SUCCESS or ERROR if the unit test does not pass successfully. The error status is logged silently and does not stops the shiny app from running by itself. The error message is logged in a log entry body. Deparsed unit test function call is logged as an event name in a log entry header.

Usage

```
log_test(..., type = "TEST", params = NULL)
```

Arguments

...	An unit test function call that is evaluated and logged.
type	A character string. A type of the event. Default for log_test is "TEST".
params	A list of additional named event-specific parameters. Default is NULL.

See Also

Other logging events functions: log_event, log_message, log_output, log_started, log_value

Examples

```
if (interactive()) {
  set_logging()
  shiny::shinyApp(
    ui = shiny::fluidPage(log_init()),
    server = function(input, output) {
      set_logging_session()
      log_test(testthat::expect_true(TRUE))
      log_test(testthat::expect_true(FALSE))
    }
  )
}
```

---

log_value *Logging value*

Description

log_value logs value of an evaluated function into log entry body and uses deparsed function call as the event name.

Usage

```
log_value(..., type = "VALUE", status = "FIRED", params = NULL)
```

Arguments

...	A function call that is evaluated, the returned value is coerced into character string, and pasted into log entry body. Deparsed function call is used as the event name in log entry header.
type	A character string. A type of the event. Default is "VALUE".
status	A character string. A status of the event. Default is "FIRED".
params	A list of additional named event-specific parameters. Default is NULL.

See Also

Other logging events functions: log_event, log_message, log_output, log_started, log_test

Examples

```
if (interactive()) {
  set_logging()
  shiny::shinyApp(
    ui = shiny::fluidPage(log_init()),
    server = function(input, output) {
      set_logging_session()
```

```
      log_value(NROW(mtcars))
    }
  )

}
```

---

purge_eventlog *Purging eventlog*

Description

purge_eventlog removes obsolete event records based on selected criteria. Please be careful. If you do not back up your eventlog, purging operation can be irreversible.

Usage

```
purge_eventlog(file = "events.log", min_build = NULL)
```

Arguments

file	A character string. Path to a file log.
min_build	An integer. Minimum build version of the app that should be kept in the eventlog after purging.

Examples

```
demo_filelog <- system.file("shiny", "demoapp/events.log",
                            package = "shinyEventLogger")

temp_file <- tempfile()
file_conn <- base::file(temp_file)
writeLines(readLines(con = demo_filelog), file_conn)
close(file_conn)

purge_eventlog(file = temp_file, min_build = 23)
```

read_eventlog *Reading eventlog*

Description

read_eventlog reads eventlog stored in a file or in a database.

Usage

```
read_eventlog(file = NULL, db = NULL, last_n = Inf, verbose = TRUE)
```

Arguments

file	A character string. Path to a file log.
db	A character string. Connection string to a mongo database.
last_n	An integer. How many last event records should be return? Default is Inf which returns the whole eventlog.
verbose	A logical value. Should the function print addition messages? Default is TRUE.

Value

An object of class eventlog which is a data frame with appropriate case, activity and timestamp classifiers specified. The eventlog object is a result of bupaR:eventlog function from bupaR package and it is suitable for further process-mining analysis.

See Also

purge_eventlog, run_demo.

Examples

```
read_eventlog(
 last_n = 25,
 file = system.file("shiny", "demoapp/events.log",
                    package = "shinyEventLogger"))
```

run_demo *Run demo shiny app*

Description

run_demo runs demo shiny app which logs different types of events. run_demo_dashboard runs demo shiny dashboard that allows for interactive analysis of events from demo app. The demo app can be also run in background and events fired in the app can be seen immediately in the demo dashboard.

Usage

```
run_demo(in_background = FALSE)

run_demo_dashboard()
```

Arguments

in_background	A logical. If TRUE the demo shiny app is run in the background on port 5555. Default is FALSE.

Functions

- run_demo: Run demo shiny app
- run_demo_dashboard: Run demo shiny dashboard

Examples

```
if (interactive()) {
  run_demo(in_background = TRUE)
  run_demo_dashboard()
}
```

set_logging *Settings for event logging*

Description

set_logging should be used outside ui and server functions, possibly in global.R, to be used only once to define where the logging should be done. Events can be sent to R console, browser JavaScript console, a eventlog file, or a database (or any combination of these). By default logging is done to the R console and JavaScript console. set_logging also can be used to define global event parameters - named objects passed to ... that will be evaluated and added to lists of parameters of all events.

Usage

```
set_logging(r_console = TRUE, js_console = TRUE, file = FALSE,
  database = FALSE, ...)
```

Arguments

r_console	A logical. Should events be logged into R console? Default is TRUE.
js_console	A logical. Should events be logged into browser JavaScript console? Default is TRUE.
file	A logical or a character string. Should events be logged to a file? Default is FALSE. If TRUE the default eventlog filename is "events.log". If character string, path and name of the filelog.
database	A logical or a character string. Should events be logged into a database? Default is FALSE. If TRUE or "mongoDB" the connection URL to the database will be read from the first line of a text file named ".db_url".
...	a set of named objects (usually of type character, numeric, or date) to be logged as parameters common to all events.

Details

set_logging assigns to the parent frame a new environment log_settings_global for storing global event parameters. If database = TRUE additional database connection object named log_db is assigned to the parent frame as well.

See Also

Other setting up logging parameters functions: log_params, set_logging_session

Examples

```
if (interactive()) {

  set_logging(r_console = TRUE,
              js_console = FALSE,
              "param_1" = 1,
              "param_2" = "A")

  shiny::shinyApp(
    ui = shiny::fluidPage(),
    server = function(input, output) {
      set_logging_session()
      log_event("Event with global params")
    }
  )
}
```

`set_logging_session` *Session-specific settings for event logging*

Description

`set_logging_session` should be used at the beginning of the shiny server function to define session-specific event parameters. `set_logging_session` also sets event counter to 1.

Usage

```
set_logging_session(...)
```

Arguments

`...`	a set of named objects (usually of type character, numeric, or date) to be logged as parameters common to all events.

Details

`set_logging_session` assigns to the parent frame new environment: `log_settings_session` for storing session-specific event parameters and information about multiple instances of the same event (see `log_started`).

See Also

Other setting up logging parameters functions: `log_params`, `set_logging`

Examples

```
if (interactive()) {

  set_logging()
  shiny::shinyApp(
    ui = shiny::fluidPage(log_init()),
    server = function(input, output) {
      set_logging_session(
        session_id = shiny::getDefaultReactiveDomain()$token
        )
      log_event("Event 1 with session parameter")
      log_event("Event 2 with session parameter")
    }
  )
  # You can open app in the browser and duplicate tab to see different
  # unique session ids in event parameters.
}
```

shinyEventLogger *shinyEventLogger: Logging Events in Shiny Apps*

Description

Logging framework dedicated for complex shiny apps. Different types of events can be logged (value of a variable, multi-line output of a function, result of a unit test, custom error, warning, or diagnostic message). Each event can be logged with a list of parameters that are event-specific, common for events within the same scope, session-specific, or app-wide. Logging can be done simultaneously to R console, browser JavaScript console, a file log, and a database (MongoDB). Log data can be further analyzed with the help of process-mining techniques from 'bupaR' package.

Author(s)

Kamil Wais <kamil.wais@gmail.com>

See Also

Useful links:

- https://github.com/kalimu/shinyEventLogger#readme
- https://kalimu.github.io/project/shinyeventlogger/
- Report bugs at https://github.com/kalimu/shinyEventLogger/issues

Index