Logging using OhdsiRTools

Martijn J. Schuemie

2018-04-23

Contents

1 Introduction

This vignette describes how you can use the `OhdsiRTools` package to perform logging. Logging is the activity of recording events that occur during an analysis in a log. The log can be used for example for for debugging, profiling (understanding performance bottlenecks), and audits.

1.1 Terminology

- **Logger**: An object that can receive **events**, and writes them to a log. A logger has a **name**, a prespecified **event level** (only events at or above that level are logged), and one or more **appenders**.
- **Event**: Consists of a message and an event level.
- **Event level**: Each event has an associated level. These levels (in ranked order) are
  - `TRACE`: Events to mark the analysis has passed through some code.
  - `DEBUG`: Events to help understand the state of the code (e.g. whether a variable has a value).
  - `INFO`: Events typically displayed to the user to inform of the progress.
  - `WARN`: Events that indicate something probably requires attention.
  - `ERROR`: Events indicating something went wrong.
  - `FATAL`: Events indicating something went wrong, causing the analysis to terminate.
- **Appender**: An object that writes to a destination, for example the console or a file. An appender uses a **layout** to format its messages. There currently are two types appenders:
  - **Console appender**: Writes to the console, created using the `createConsoleAppender` function.
  - **File appender**: Writes to a file, created using the `createFileAppender` function.
- **Layout**: Objects specifying the format in which the log will be created. The following layouts are available:
  - `layoutSimple`: Only outputs the message.
  - `layoutTimestamp`: Adds the current time and date to the message.
  - `layoutStackTrace`: Adds the time and date, and full stack trace to the message.
  - `layoutParallel`: Includes the thread identifier, name of the package and function raising the event, the current time and date, the message level, and the message itself.

2 Creating a console logger

The code below demonstrates how one would create a logger that writes all events at level `INFO` or greater to the console using a layout with time stamp:

```
logger <- createLogger(name = "SIMPLE",
                       threshold = "INFO",
                       appenders = list(createConsoleAppender(layout = layoutTimestamp)))

registerLogger(logger)

logTrace("This event is below the threshold (INFO)")

logInfo("Hello world")
```

```
#> 2018-04-23 15:12:38  Hello world
```

Note that the message is displayed twice. This is because there is a default logger that uses the simple layout and threshold = "INFO", and writes to console. We can remove this logger before registering our logger to avoid duplication:

```
clearLoggers()

logger <- createLogger(name = "SIMPLE",
                       threshold = "INFO",
                       appenders = list(createConsoleAppender(layout = layoutTimestamp)))

registerLogger(logger)

logInfo("Hello world")
```

```
#> 2018-04-23 15:12:38  Hello world
```

3 Creating a file logger

Probably more useful is a file logger. In the code below, we instantiate a logger that writes to file, using a threshold of `TRACE` (so including all events), and using the layout for parallel processing.

```
logger <- createLogger(name = "PARALLEL",
                       threshold = "TRACE",
                       appenders = list(createFileAppender(layout = layoutParallel,
                                                           fileName = "log.txt")))
registerLogger(logger)

logTrace("Executed this line")

logDebug("There are ",  length(getLoggers()), " loggers")

logInfo("Hello world")
```

```
#> 2018-04-23 15:12:38  Hello world
```

We can read the log file:

```
writeLines(readChar("log.txt", file.info("log.txt")$size))
```

```
#> 2018-04-23 15:12:38  [Main thread]   TRACE   evaluate    timing_fn   Executed this line
#> 2018-04-23 15:12:38  [Main thread]   DEBUG   evaluate    timing_fn   There are 2 loggers
#> 2018-04-23 15:12:38  [Main thread]   INFO    evaluate    timing_fn   Hello world
```

And clean it up when we're done:

```
unlink("log.txt")
```

3.1 Shorthand

A shorthand for creating the file logger detailed here is offered by the `addDefaultFileLogger` function. The code

```
addDefaultFileLogger("log.txt")
```

is equivalent to

```
registerLogger(createLogger(name = "DEFAULT",
                            threshold = "TRACE",
                            appenders = list(createFileAppender(layout = layoutParallel,
                                                                fileName = "log.txt"))))
```

4 Warnings and fatal errors

All R warnings and errors are automatically logged, and therefore do not require explicit logging. For example:

```
clearLoggers()
addDefaultFileLogger("log.txt")

warning("Danger!")

# This throws a warning:
as.numeric('a')

# This throws an error:
a <- b

writeLines(readChar("log.txt", file.info("log.txt")$size))
```

```
#> 2018-04-23 15:12:38  [Main thread]   WARN    evaluate    timing_fn   Danger!
#> 2018-04-23 15:12:38  [Main thread]   WARN    evaluate    timing_fn   Warning: NAs introduced by coer
#> 2018-04-23 15:12:38  [Main thread]   FATAL   evaluate    timing_fn   Error: object a not found
```

5 Logging when parallel processing

The logging functions are designed to work with the parallel processing functions included in this package. The `layoutParallel` records thread identifiers, making it possible to later untangle the various threads. Below is a simple example:

```
unlink("log.txt") # Clean up log file from the previous example
clearLoggers() # Clean up the loggers from the previous example

addDefaultFileLogger("log.txt")
```

```
cluster <- makeCluster(3)

fun <- function(x) {
  OhdsiRTools::logInfo("The value of x is ", x)
  # Do something
  if (x == 6)
    OhdsiRTools::logDebug("X equals 6")
  return(NULL)
}

dummy <- clusterApply(cluster, 1:10, fun, progressBar = FALSE)

stopCluster(cluster)

writeLines(readChar("log.txt", file.info("log.txt")$size))
```

```
#> 2018-04-23 15:12:38  [Main thread]   TRACE   evaluate    timing_fn   Initiating cluster with 3 threa
#> 2018-04-23 15:12:48  [Thread 1]  TRACE           Thread 1 initiated
#> 2018-04-23 15:12:48  [Thread 2]  TRACE           Thread 2 initiated
#> 2018-04-23 15:12:48  [Thread 3]  TRACE           Thread 3 initiated
#> 2018-04-23 15:12:48  [Thread 1]  INFO            The value of x is 1
#> 2018-04-23 15:12:48  [Thread 2]  INFO            The value of x is 2
#> 2018-04-23 15:12:48  [Thread 3]  INFO            The value of x is 3
#> 2018-04-23 15:12:52  [Thread 2]  INFO            The value of x is 5
#> 2018-04-23 15:12:52  [Thread 3]  INFO            The value of x is 6
#> 2018-04-23 15:12:52  [Thread 3]  DEBUG           X equals 6
#> 2018-04-23 15:12:52  [Thread 1]  INFO            The value of x is 7
#> 2018-04-23 15:12:52  [Thread 1]  INFO            The value of x is 10
#> 2018-04-23 15:12:52  [Thread 2]  INFO            The value of x is 8
#> 2018-04-23 15:12:52  [Thread 3]  INFO            The value of x is 9
#> 2018-04-23 15:12:52  [Main thread]   TRACE   evaluate    timing_fn   Stopping cluster
#> 2018-04-23 15:12:52  [Thread 1]  TRACE           Thread 1 terminated
#> 2018-04-23 15:12:52  [Thread 3]  TRACE           Thread 3 terminated
```

6 Shiny log viewer

A Shiny app for viewing a log file created using the `layoutParallel` is included in the package. To explore the log created in the prior example, run

```
launchLogViewer("log.txt")
```

to launch the viewer shown in Figure 1.

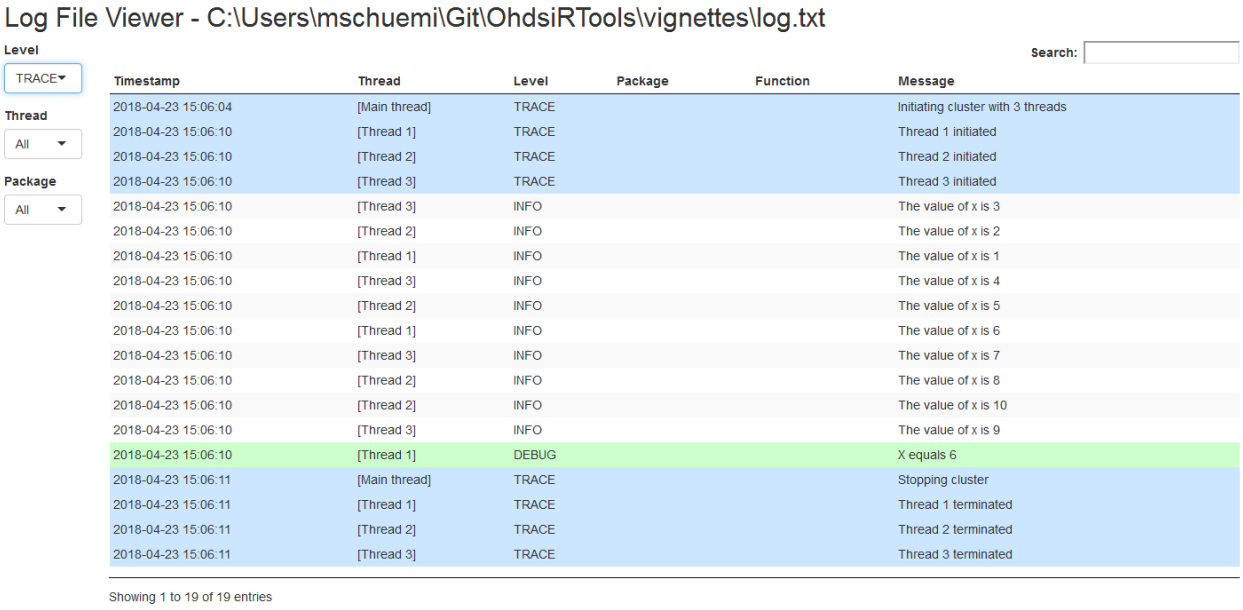

Figure 1: Shiny log viewer app