The [plumber R package](https://www.rplumber.io/docs/) is used to expose R functions as API endpoints. Due to plumber’s incredible flexibility, most major API design decisions are left up to the developer. One important consideration to be made when developing APIs is how to log information about API requests and responses. This information can be used to determine how plumber APIs are performing and how they are being utilized.

An example of logging API requests in plumber is included in the [package documentation](https://www.rplumber.io/docs/routing-and-input.html#filters). That example uses a filter to log information about incoming requests before a response has been generated. This is certainly a valid approach, but it means that the log cannot contain details about the response since the response hasn’t been created yet. In this post we will look at an alternative approach to logging plumber APIs that uses [preroute and postroute hooks](https://www.rplumber.io/docs/programmatic-usage.html#router-hooks) to log information about each API request and its associated response.

**Logging**

In this example, we will use the [logger package](https://daroczig.github.io/logger/) to generate the actual log entries. Using this package isn’t required, but it does provide some convenient functionality that we will take advantage of.

Since we will be registering hooks for our API, we will need both a plumber.R file and an entrypoint.R file. The plumber.R file contains the following:

# plumber.R

# A simple API to illustrate logging with Plumber

library(plumber)

#\* @apiTitle Logging Example

#\* @apiDescription Simple example API for implementing logging with Plumber

#\* Echo back the input

#\* @param msg The message to echo

#\* @get /echo

function(msg = "") {

list(msg = paste0("The message is: '", msg, "'"))

}

#\* Plot a histogram

#\* @png

#\* @get /plot

function() {

rand <- rnorm(100)

hist(rand)

}

Now that we’ve defined two endpoints (/echo and /plot), we can use entrypoint.R to setup logging using preroute and postroute hooks. First, we need to configure the logger package:

# entrypoint.R

library(plumber)

# logging

library(logger)

# Specify how logs are written

log\_dir <- "logs"

if (!fs::dir\_exists(log\_dir)) fs::dir\_create(log\_dir)

log\_appender(appender\_tee(tempfile("plumber\_", log\_dir, ".log")))

The log\_appender() function is used to specify which appender method is used for logging. Here we use appender\_tee() so that logs will be written to stdout and to a specific file path. We create a directory called logs/ in the current working directory to store the resulting logs. Every log file is assigned a unique name using tempfile(). This prevents errors that can occur if concurrent processes try to write to the same file.

Now, we need to create a helper function that we will use when creating log entries:

convert\_empty <- function(string) {

if (string == "") {

"-"

} else {

string

}

}

This function takes an empty string and converts it into a dash ("-"). We will use this to ensure that empty log values still get recorded so that it is easy to read the log files. We’re now ready to create our plumber router and register the hooks necessary for logging:

pr <- plumb("plumber.R")

pr$registerHooks(

list(

preroute = function() {

# Start timer for log info

tictoc::tic()

},

postroute = function(req, res) {

end <- tictoc::toc(quiet = TRUE)

# Log details about the request and the response

log\_info('{convert\_empty(req$REMOTE\_ADDR)} "{convert\_empty(req$HTTP\_USER\_AGENT)}" {convert\_empty(req$HTTP\_HOST)} {convert\_empty(req$REQUEST\_METHOD)} {convert\_empty(req$PATH\_INFO)} {convert\_empty(res$status)} {round(end$toc - end$tic, digits = getOption("digits", 5))}')

}

)

)

pr

We use the $registerHooks() method to register both preroute and postroute hooks. The preroute hook uses the [tictoc package](http://collectivemedia.github.io/tictoc/) to start a timer. The postroute hook stops the timer and then writes a log entry using the log\_info() function from the logger package. Each log entry contains the following information:

* Log level: This is a distinction made by the logger package, and in this  
  example the value is always INFO
* Timestamp: The timestamp for when the response was generated and sent back to  
  the client
* Remote Address: The address of the client making the request
* User Agent: The user agent making the request
* Http Host: The host of the API
* Method: The HTTP method attached to the request
* Path: The specific API endpoint requested
* Status: The HTTP status of the response
* Execution Time: The amount of time from when the request received until the response was generated

This log format is loosely inspired by the [NCSA Common log format](https://en.wikipedia.org/wiki/Common_Log_Format).

**Testing**

Now that our API is all setup, it’s time to test to make sure logging works as expected. First, we need to start the API. The easiest way to do this is to click the Run API button that appears at the top of the plumber.R file in the RStudio IDE. Once the API is running, you’ll see a message in the console similar to the following:

Running plumber API at <http://127.0.0.1:5762>

Now that we know the API is running, we need to make a request. One of the easiest ways to make a request in this case is to open a web browser (like Google Chrome) and type the API address in the address bar followed by /plot. In this example, I would type <http://127.0.0.1:5762/plot> into the address bar of my browser. If all goes well, you should see a plot rendered in the browser. The RStudio console will display the log output:

INFO [2019-08-09 12:30:23] 127.0.0.1 "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_14\_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/76.0.3809.100 Safari/537.36" localhost:5762 GET /plot 200 0.158

A new logs/ directory will have been created in the current working directory and it will contain a file with the log entry. You can generate more log entries by refreshing your browser window.

**Analyzing**

Let’s say that we refreshed the browser window 1,000 times. The log file generated will contain an entry for each request. We can analyze this log file to find helpful information about the API. For example, we could plot a histogram of execution time:

library(ggplot2)

plumber\_log <- readr::read\_log("logs/plumber\_fe3daed895d.log",

col\_names = c("log\_level",

"timestamp",

"remote\_address",

"user\_agent",

"http\_host",

"method",

"path",

"status",

"execution\_time"))

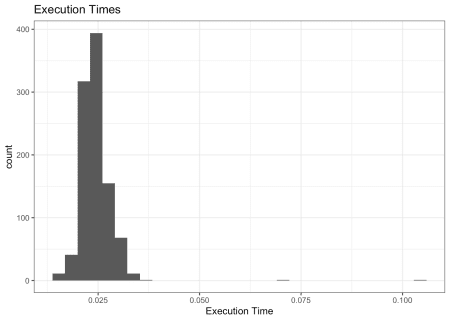
ggplot(plumber\_log, aes(x = execution\_time)) +

geom\_histogram() +

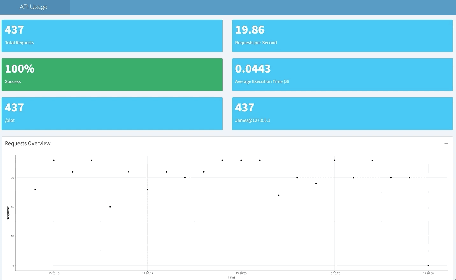
theme\_bw() +

labs(title = "Execution Times",

x = "Execution Time")



We could even build a [Shiny application](http://shiny.rstudio.com/) to monitor the logs/ directory and provide real-time visibility into API metrics!



The details of this Shiny application go beyond the scope of this post, but the source code is available [here](https://github.com/sol-eng/plumber-logging/blob/master/R/shiny/app.R).

Plumber APIs published to [RStudio Connect](https://www.rstudio.com/products/connect/) can use this pattern to log and monitor API requests. Details on this use case can be found in [this repository](https://github.com/sol-eng/plumber-logging#deployment)

**Conclusion**

Plumber is an incredibly flexible package for exposing R functions as API endpoints. Logging information about API requests and responses provides visibility into API usage and performance. These log files can be manually inspected or used in connection with other tools (like Shiny) to provide real-time metrics around API use.

If you are interested in learning more about using plumber, logging, Shiny, and RStudio Connect, please visit [community.rstudio.com](https://rviews.rstudio.com/2019/08/13/plumber-logging/community.rstudio.com) and let us know!