PharmaSUG 2023 - Paper AP-063

RESTful Thinking:

Using R Shiny and Python to streamline REST API requests and visualize REST API responses

Laura Elliott, MPH, SAS ® Institute Inc.;
Crystal Cheng, MS, SAS ® Institute Inc.

ABSTRACT

REST APIs are a popular way to make HTTP requests to access and use data due to their simplicity. They can be used to carry out several types of actions in a statistical computing environment. Even though they are simple, some limitations have been observed such as lack of detail in responses, difficulty in debugging the failure of certain actions, and execution requires the user to have some basic HTTP request knowledge. This research focuses on mitigating these limitations by utilizing the strengths of both R and Python to build a user interface that executes REST APIs easily and displays responses with more detail. R Shiny was used to create an easy-to-use interface that contains several embedded HTTP requests, written using the Python requests module, that can be easily executed regardless of a user's previous knowledge. These requests perform specified actions in a statistical computing environment and return detailed results to be viewable in the R Shiny dashboard. This paper will explain the concept and build process of the dashboard, will discuss techniques used to integrate R and Python programming languages, and will introduce the resulting dashboard. In the end, the paper will discuss challenges faced during development and some considerations for the future enhancement of REST APIs. The products used for development include R and Python programming languages and the statistical computing environment SAS Life Sciences Analytics Framework. This paper is intended for individuals with R and Python experience, and those who have knowledge of REST APIs.

Keywords: REST API, RStudio, Python, integration, R Shiny, dashboard, SAS ®, reticulate, HTTP request, user-friendly, SAS® Life Sciences Analytics Framework

INTRODUCTION

A REST API is a specific type of HTTP request that allows applications' clients and servers to communicate. REST stands for Representational State Transfer, which is a software architectural style that consists of a set of design principles: a uniform interface, a client-server, is stateless, is cacheable, is a layered system, and code is on demand. REST APIs are flexible to implement and easy to understand, they are useful for quick data exchange with servers and clients on web applications. Users can leverage REST APIs to create, retrieve, delete, or update records within the resource by utilizing POST, GET, PUT, DELETE request accordingly. The responses to requests are sent back to clients in different data formats, such as JSON, HTML, Plain text, etc. Due to the flexibility of the REST APIs, many applications have adopted them. With REST APIs, the process of applications integration and data sharing is simplified. Despite this simplicity, however; there are still some observed limitations of REST APIs when using them to make HTTP requests in a statistical computing environment. Some limitations observed include responses not being descriptive enough when running a program as a job, failure of a request is difficult to debug, and running an HTTP request requires some previous experience in programming. This paper discusses the development of a dashboard using the R Shiny package available in RStudio. The dashboard attempts to enhance REST API execution and response interpretation by utilizing embedded Python and R API functions to communicate with the statistical computing environment SAS ® Life Sciences Analytics Framework 5.4.1. Readers of this paper will gain knowledge of how to use Python and RStudio together to build tools that could streamline otherwise manual processes, will gain some insight in coding REST APIs in Python and RStudio, and will be introduced to some of the many prospective functionalities a tool of this type can possess. While development of a tool like this requires some previous Python, RStudio, and HTTP request knowledge, using the REST API dashboard is suitable for users of any

skill level.

REST API DASHBOARD OVERVIEW

The REST API dashboard for this statistical computing environment contains three major components: the R Shiny user interface, REST API codes, and the responses returned by the statistical computing environment. Refer to Figure 1 for a high-level overview of how the tool operates.

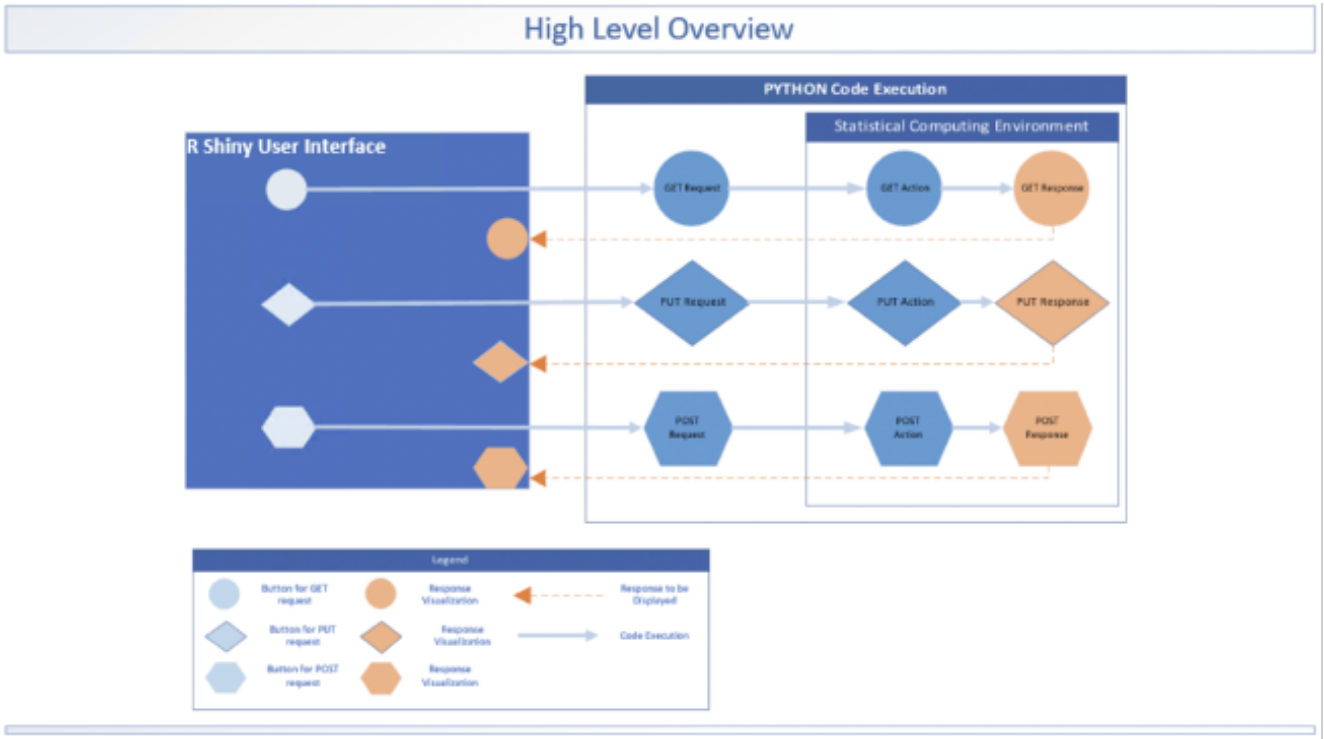


Figure 1. High Level Overview

The R Shiny interface allows for easy execution of each REST API with a click of a button represented by the light blue shapes. Different shapes represent a different request type i.e. GET(circle), PUT(diamond), and POST(hexagon). When a button is selected it triggers the REST API code on the back end, which is written in either R or Python. The REST API completes an action in the statistical computing environment and easy-to-interpret responses are then displayed back on the R Shiny dashboard, represented by the orange shapes.

DASHBOARD DEVELOPMENT

STATISTICAL COMPUTING ENVIRONMENT

The statistical computing environment in this research is the most recent version of SAS ® Institute's Life Sciences Analytics Framework, version 5.4.1. SAS ® Life Sciences Analytics Framework is a collaborative cloud-based environment used for clinical research. This environment contains a repository area, where master copies are stored as "read-only" which can be accessed by users who have the correct permissions, and a workspace, which is a private user area where items from the repository can be manipulated, and code can be developed. Items that are manipulated or developed in the workspace can be synced back to the repository so that the most current updates to a particular item are available to other users with correct access.

Users can utilize REST APIs to carry out various actions of both the repository and the workspace areas. There are several REST APIs available for both repository and the workspace including uploading and

downloading content, getting item properties, and running SAS ® programs through a job.

REST API METHODS TESTED

Common REST API methods available are GET requests and PUT requests. A GET request is used to request data from a specified source, while a PUT request is used to send data to a server to create or update a resource (W3Schools, 2023).

There are several GET and PUT methods available for this statistical computing environment to test. While developing requests, there is a basic code structure for both request types in RStudio and Python and structure for embedding them into R Shiny. Just a few minor changes are necessary based on the action desired by the user. The GET and PUT requests in the dashboard were coded in both RStudio and Python languages to observe the strengths and weaknesses, if any, of each.

RSTUDIO AND PYTHON INTEGRATION

Package Requirements

Python version 3.11.2 and RStudio version 4.2.3 were used to program the REST API calls and build the user interface. To ensure smooth development, it is crucial to download the R version that works with the operating system. For example, if a developer is using macOS Big Sur or higher with and Apple Silicon arm64, the R version for this type of system needs to be downloaded. The required Python packages are the 'requests' module that allows the program to make HTTP requests and the 'os' module which allows the program to access the local operating system for requests like uploading and downloading content to a local directory. To achieve interoperability between Python and RStudio, the R package 'reticulate' is required. The 'reticulate' package allows the programmer to use Python within an RStudio session and allows for translation between R and Python objects (Shiny. 2023). Several other RStudio packages were used to build this dashboard, however, the most important in addition to 'reticulate' is 'HTTR.' The HTTR package is required to make HTTP requests in the R language. With HTTR package installed, a user can use HTTP verbs such as GET, POST, PUT to send the HTTP requests.

Python

Python HTTP Functions

To be utilized by the reticulate package, each HTTP request coded in Python is written as a function with the necessary arguments to carry out the request. The functions are stored in a Python directory within the R Shiny application's working directory. These functions are easily embedded and called by R Shiny. The below code snippet shows an example of a Python function, 'get_fileprop_ws' that executes a GET request to receive a specified file's properties. The arguments necessary to carry out the API call are the statistical computing environment's HTTP path, the file path in the environment, and the authentication token needed to make any REST API request in the statistical computing environment. The required arguments are identified within the SAS® Life Sciences Analytics Framework REST API Documentation.

Python function get_fileprop_ws:

```
def get_fileprop_ws(path, wsfilepath, token):
  import json
  import requests

  path2 = wsfilepath.replace(" ","%20")

  lsaf_getprop = (path + "/lsaf/api/workspace/files" + path2 +
"?component=properties")
  headers = {'X-Auth-Token': token}
```

```
#make request
getprop = requests.get(lsaf_getprop, headers=headers)

#return message
prop_stat = str(getprop.status_code)
properties = getprop.json()
return(properties)
```

Embedding Python Functions in R Shiny

Embedding a Python function in R Shiny involves 3 basic steps. These general steps are consistent across request types, only minor changes are needed based on the arguments identified within the REST API documentation. The following code examples show the steps necessary to embed the Python function 'get_fileprop_ws' in the R Shiny dashboard. Refer to Display 1 to see this example in the dashboard view.

1. The arguments in the Python function are declared as user inputs in the R Shiny UI code. In this example, the path to a specific file in the statistical computing environment is stored as the user input "pathname_fpws" :

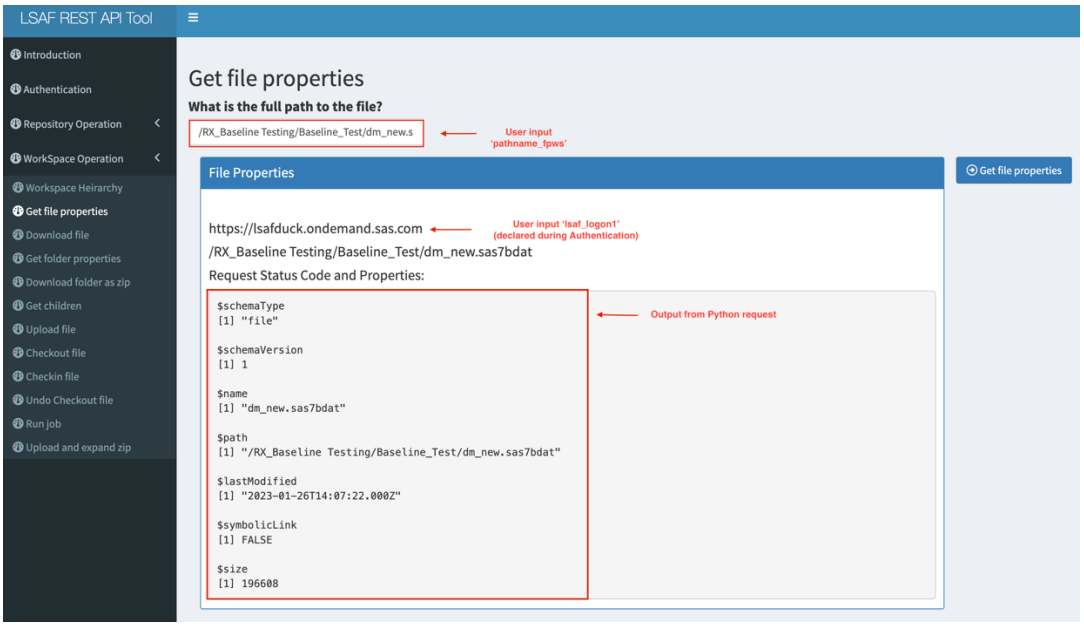
```
tabItem(tabName = "wstab1"
         ,h2("Get file properties")
         ,h4(textInput("pathname_fpws","What is the full path to the
         file?"))
         ,actionButton(inputId = "getwsprop",label="Get file
         properties",icon=icon("arrow-alt-circle-right"), style="color:
         #fff; background-color: #337ab7; border-color: #2e6da4")
```

2. In the R Shiny server code, the python function is executed using 'reticulate' when a button is clicked. This code snippet shows how the user input from the previous step 'pathname_fpws' is utilized as the second argument 'wsfilepath' in the Python function 'get_fileprop_ws' (other arguments 'lsaf_logon1' and 'x' declared during authentication, are required for all requests):

```
wsgetfileprop <- eventReactive(input$getwsprop, {
   lsaf_getfilepropws <-
   reticulate::source_python("Python_WS/GET_fileprop_ws.py")
   get_fp <- get_fileprop_ws(input$lsaf_logon1,input$pathname_fpws,
   token$x)
```

3. Once the function is executed and request is complete, HTTP responses are easily displayed, by using 'return' command in the server code, in JSON format. R Shiny outputs are declared in the R Shiny UI code:

```
output$filepropstatus <- renderPrint({return(get_fp)}
```



Display 1. Dashboard view of request 'get_fileprop_ws'

The dashboard view shows how each component displays on the dashboard after the action button is clicked to execute a Python function.

RStudio

RStudio HTTP Requests

Making requests using the R language is similar to Python. The major difference is that the R requests can be coded in the R language and inserted into the R Shiny apps server code whereas Python functions are stored outside of the application in the working directory. Other differences observed included the package needed to make requests and the ease of converting responses into an R data frame.

The following R Shiny server code example displays the same GET request as in the Python example above written in the R language. Using the 'HTTR' RStudio package, the code sends a HTTP request to the statistical computing environment and retrieves the response in JSON format. In order to better visualize the response, the JSON file is converted to a data frame and displayed on the R Shiny dashboard. Display 2 shows the following in the dashboard view:

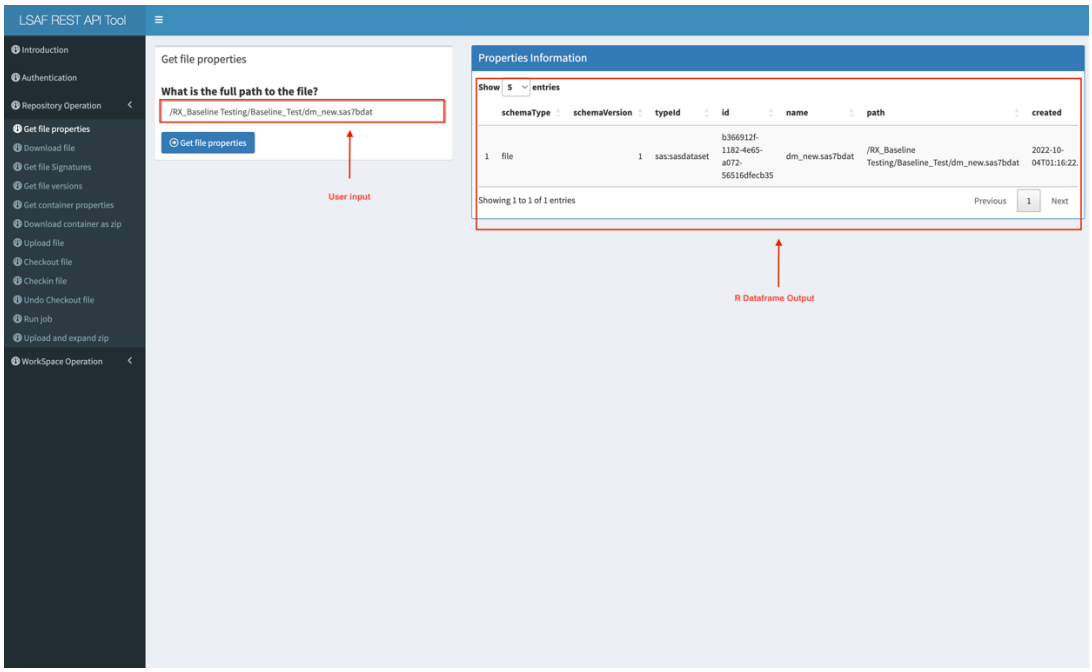
```
getfileprop <- eventReactive(input$getprop,{
      pathname <- gsub(" ","%20",input$pathname)
      lsaf_getprop <-
      paste0(input$lsaf_logon1,"/lsaf/api/repository/files",pathname,"?compon
      ent=properties")

getprop <- httr::GET(lsaf_getprop,add_headers("x-auth-token" =
token$x),content_type("application/json"))
getproplst <- content(getprop)

if (getproplst[["versioned"]]=="TRUE") {
      clns <- as.data.frame(getproplst[1:(length(getproplst)-1)])
      extraclns<-as.data.frame(do.call(rbind,getproplst[length(getproplst)]))
      rownames(extraclns)<-NULL
      propdf$df<- cbind(clns,extraclns)

  } else {
```

```
    propdf$df <- as.data.frame(getproplst)
}
```



Display 2. Dashboard view of R Studio request

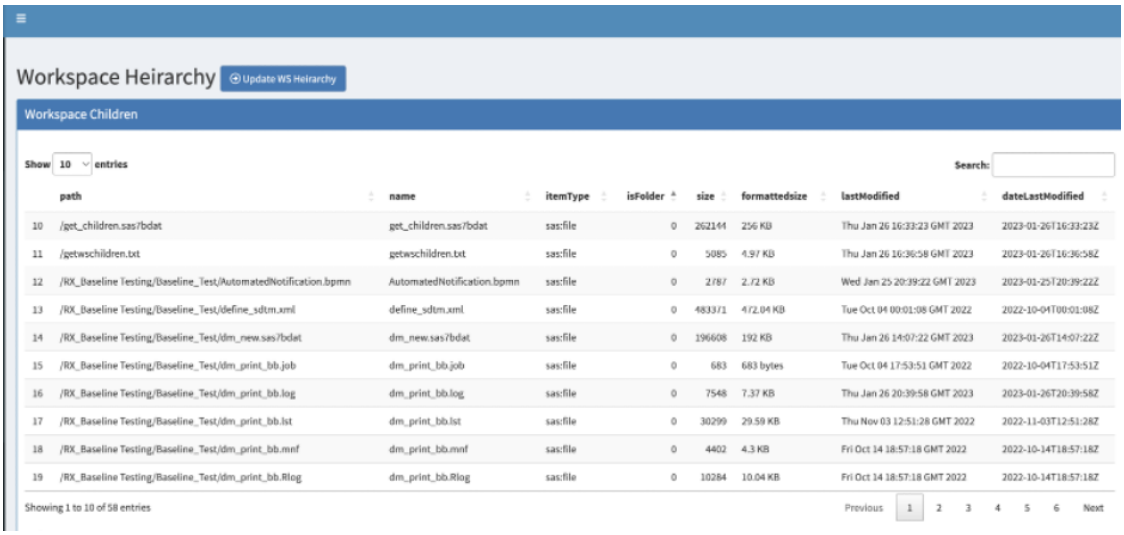
The dashboard view shows how each component displays on the dashboard after the action button is clicked to execute a R Studio HTTP request and shows the R data frame output.

COMBINED REQUESTS

Hierarchy Information

An advantage of using a REST API dashboard is the ability to execute multiple requests when one action occurs. One of the goals of this dashboard is to use it without the need to sign into the statistical computing environment's URL to get paths to files and folders. To accomplish this, the below example shows a use of a combined request containing the "run job" API and "download file" API embedded in a section of the dashboard. These APIs are stored as one reactive event that displays either the workspace or repository hierarchy when the "update" action button is clicked. This data contains the paths for all items in SAS® Life Sciences Analytics Framework which gives the user the ability to use the REST API dashboard without having to log into the statistical computing environment's user interface.

When the button is clicked, the "run job" API executes to run a job that creates a dataset that contains all the contents in the repository or workspace. The data is updated every time the action button is clicked. Display 3 shows the dashboard view of the combined request after it has been executed.



Display 3. Dashboard view of Hierarchy Information Combined Request

The dashboard view displays the hierarchy information within the statistical computing environment after the action button is clicked.

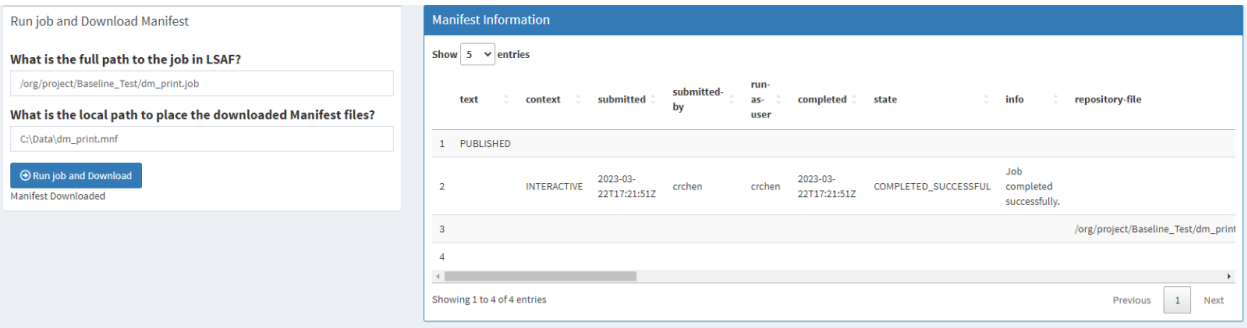
Running Jobs

Combined requests prove useful for running jobs. When running a job by itself using the CURL command, a lack of detail in the response leaves the user wondering if the job finished successfully. The CURL response only lets a user know if a job has started. Using the same combination of REST APIs, “run job” + “download file,” the log and/or manifest created by a job can be downloaded automatically to display a little more detail about a job’s completion status. Display 4 shows the response from a “run job” request when executed by itself in the command line while Display 5 exhibits utilization of both “run job” and “download” APIs using the dashboard to display more detail about the status of the job.


```
{
   "schemaType": "RepositoryExecuteActionResult",
   "schemaVersion": 1,
   "action": "run",
 - "status": {
      "schemaVersion": 1,
      "code": 40001,
      "type": "started"
   },
 - "item": {
      "schemaType": "executable",
      "schemaVersion": 1,
      "path": "/org/project/jobs/test.job"
   },
   "submissionId": "a492ceff-d529-4eb9-aa08-cd368fbf1960"
}
```


Display 4. Run job API response from Command Line

The command line response when the run job API is executed. This response is useful in telling the user if a job has started or not but does not give any information about whether or not the job completed successfully.



Display 5. Combined Request dashboard view

The dashboard runs a job and downloads the manifest. The manifest contents display the job completion status in the 'state' and 'info' columns.

CONCLUSION

The REST API dashboard mitigates some limitations observed while running them in the command line tool. Embedding the REST APIs in an R Shiny dashboard enables the user to carry out quick and easy data exchange with a statistical computing environment. Responses are returned from users' HTTP requests and displayed in a more user-friendly way. The REST API tool can be tailored to the end users' needs. Users can embed other platforms' APIs to add more functionalities to the tool. This tool could improve the efficiency of a user by giving them the ability to complete various tasks across multiple platforms in one place. Potential future enhancements could include incorporating clinical data visualization and data summary reporting features.

REFERENCES

IBM. "What is a REST API?"

https://www.ibm.com/topics/rest-apis

Ranikay. 2021. "Shiny-reticulate-app." Accessed January 2023.

https://github.com/ranikay/shiny-reticulate-app/blob/master/server.R

Shiny. 2023. "Function reference." Accessed January 2023.

https://shiny.rstudio.com/reference/shiny/0.11/

W3Schools. 2023. "HTTP Request Methods." Accessed March 1, 2023. https://www.w3schools.com/tags/ref_httpmethods.asp.

Wikipedia. "Representational state transfer"

https://en.wikipedia.org/wiki/Representational_state_transfer

HELPFUL RESOURCES

To better understand the 'reticulate' package the following resources are useful:

- *"Shiny – reticulate – App":* *https://github.com/ranikay/shiny-reticulate-app/blob/master/server.R*
- *"R interface to Python":* *https://rstudio.github.io/reticulate/*

To better understand the functions available in R Shiny, the following in useful:

- https://shiny.rstudio.com/reference/shiny/0.11/

CONTACT INFORMATION

Comments and suggestions may be sent to:

Laura Elliott, MPH
Industry Consultant
SAS ® Institute Inc.
laura.elliott@sas.com

Crystal Cheng, MS
Industry Consultant
SAS ® Institute Inc.
crystal.cheng@sas.com

TRADEMARK CITATIONS