CT02

A Beginner's Guide to a Multilingual Shiny App

Karl Kennedy, GSK, London, UK

ABSTRACT

Within the Shiny Environment you can leverage the power of multiple languages/technologies e.g., R (obviously!), JavaScript, Python, CSS, and R Markdown. The primary goal of this paper is to demonstrate the usage of these relatively advanced topics in a simple Shiny app so Shiny beginners can utilise and incorporate these concepts into their first Shiny apps. While also reducing the programming learning curve in implementing these new concepts. It will use the power of Python's string operations, CSS and JavaScript for front-end functionality, and R Markdown for reporting results. It will also cover some basic Shiny fundamentals along with good programming methodologies e.g., modularisation.

INTRODUCTION

The motivation for this paper is to share a few tips that I hope will reduce the Shiny learning curve time so fellow novices can produce more efficient and clean apps and avoid “spaghetti” coding. If I was aware of how to implement modularisation for my first Shiny app, I could have reduced the development time while also improving maintainability of the app. I am almost certain this would have reduced my colleagues' frustrations when they had to make updates to the app.

Since this paper is aimed at Shiny novices like myself, I will start off by providing a high-level introduction to setting up a Shiny app before describing each concept in turn. I will use a basic Shiny app that I developed to demonstrate the concepts described in the paper.

SHINY SET-UP

Shiny applications consist of two core components, the user interface (UI) and the server component. These are passed as arguments to the shinyApp function that creates a Shiny app object from this UI/server pair [1].

There are two approaches to configuring/organising your Shiny core program files. These are:

- app.R - which contains UI and Server components in one file.
- Two separate files for the UI (ui.R) and the server (server.R) components.

My preference is to separate the UI and server components into two separate files and use global.R for my global variables. Below are the outlines of the UI and server functions within ui.R and server.R.

```
ui <- fluidPage(
# Place your UI components

)

server <- function(input, output) {
 # Place your server/application logic

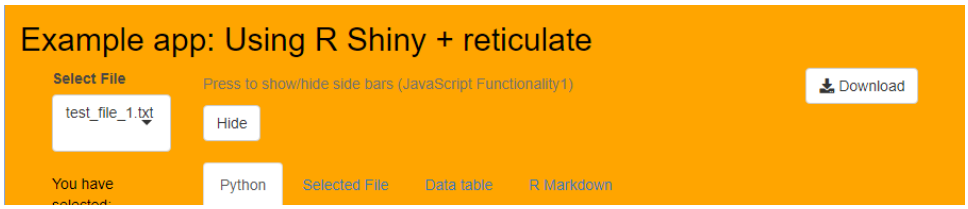
 }
```

JAVASCRIPT AND CSS

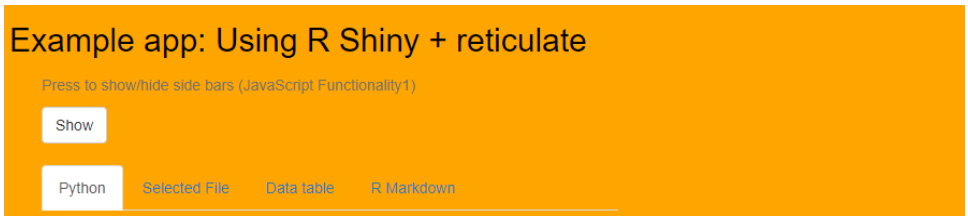
In my Shiny app I used JavaScript to implement the functionality of displaying and hiding the side bars of the app. To use JavaScript functionality, you need to use the ‘shinyjs’ package.

The basic layout of the frontend of the app are two sidebars and a main section. Each side bar has a width of two columns and the main section has a width of 8 columns. The show and hide functionality work by using the ‘showElement’ and ‘hideElement’ functions from shinyjs and uses a communication mechanism to send messages from R to JavaScript to expand or reduce the width of the main section of the app.

The app with the sidebars is displayed below.



When the 'Hide' button is clicked the sidebars are hidden.



To send messages between R and Javascript you send messages through the method 'session$sendCustomMessage(type, message)' on the session object.

This is a code snippet from the 'Show/Hide' event listener which is in server.R. The custom message is telling Javascript to hide the two sidebars.

```
if(btnlbl == hidebtnlbl){
  # hideElement is a function from shinyJS package.
  hideElement(selector = sidebar1)
  hideElement(selector = sidebar2)

  session$sendCustomMessage("hide", hide(tblid))
}

# This function hide contains the page section id that needs to be resized after
# the sidebars have been hidden.
hide <- function(tblid){
  tblid
}
```

To receive messages on the JavaScript side you need to set up the function 'addCustomMessageHandler' in ui.R.

```
 tags$head(
    # Listener for show and hide sidebars.
    tags$script("Shiny.addCustomMessageHandler('hide', function(main){
        const input = document.getElementByClassName('col-sm-8 ' + main);
        input[0].className = 'col-sm-12 ' + main
    });"),

    tags$script("Shiny.addCustomMessageHandler('show', function(main){
         const input = document.getElementByClassName('col-sm-12 ' + main);
         input[0].className = 'col-sm-8 ' + main
   });"),
```

These message handlers use a HTML DOM function to get the class name of the main section of the page and expand the width to the maximum column width of 12 or reduce the size back to 8 if you want to display the sidebars. Reference [2] provides an excellent article on how to use JavaScript within Shiny.

The HTML function enables you to use CSS. This code is stored in the ui.R and sets the background colour to orange. Depending upon your comfort level with CSS this is great addition to your Shiny arsenal to improve the app's look and feel.

```
tags$style(HTML("
  body {
    background-color: orange;
    color: black;
  }
  h2 {
    font-family: 'Yusei Magic', sans-serif;
  }
 )
```

PYTHON

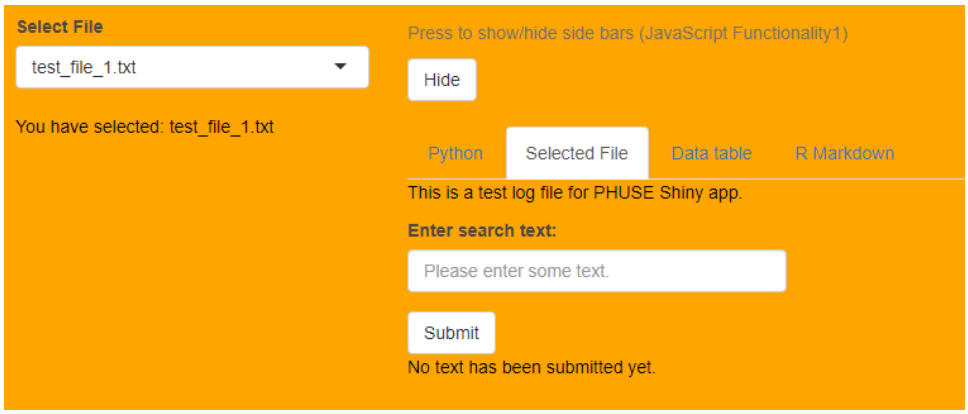
The reticulate package enables the usage of Python within R. Reference [3] is a link to an excellent GitHub repository that provides a tutorial to how to configure reticulate virtual environments for use locally and for publishing to RSConnect. After following the reticulate tutorial, I was able to display my Python system information like the example in [3].

An example of a reticulate function to display the python version.

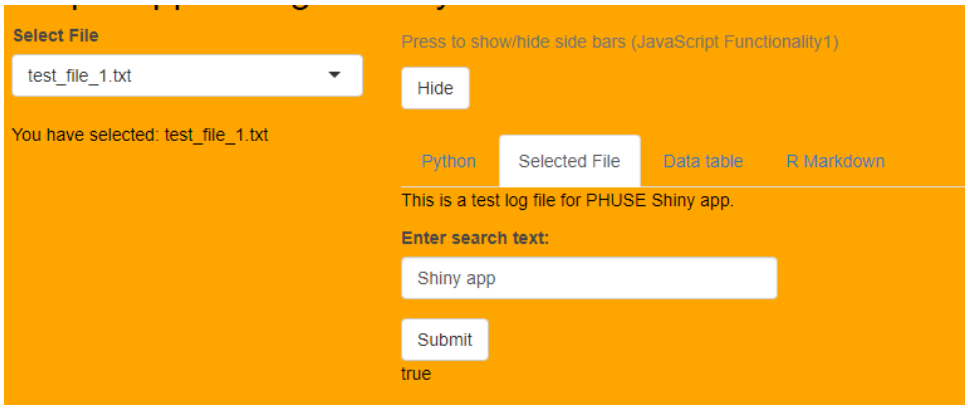
```
# Display Python version
output$python_version <- renderText({
  rr = reticulate::py_discover_config(use_environment = 'python35_env')

   paste0('Python version: ', rr$version)
})
```

Another piece of Python functionality used in the app is to perform a search of a text file.



The user selects a text file from a drop-down menu and the text contained in the file is displayed under the ‘Selected File’ tab. To search for specific text within the file, enter the search text within and search text field and press ‘Submit’. If the search text is present, then the boolean value ‘true’ is displayed on the app.



To link your Python functions to R use the reticulate function 'source_python'.

```
# Import python functions to R
reticulate::source_python('python_functions.py')
```

Within the 'python_functions' script I placed all my functions that I want to import into R.

```
def check_file(fn, txt):
  with open(fn) as f:
    x = "false"
    if txt in f.read():
      x = "true"
  return(x)
```

The above Python function searches a text file for specific text and returns either true or false if the file contains the search text. This function is executed when the submit button is pressed. This code is the event listener for the submit button. The fn parameter is path to the text file and txt parameter contains the search text.

```
observeEvent(input$submit,
{ text_reactive$text = check_file(fn=paste0("logs/", input$selectfile), txt=
                                  input$user_text)
})
```

MODULARISATION

References [4] and [5] are two excellent articles about implementing Shiny modularisation and avoiding that spaghetti coding! Unfortunately providing a detailed explanation of modularisation is beyond the scope of this paper.

"A Shiny module is a piece of a Shiny app. It can't be directly run, as a Shiny app can. Instead, it is included as part of a larger app. Modules can represent input, output, or both. They can be as simple as a single output, or as complicated as a multi-tabbed interface festooned with controls/outputs driven by multiple reactive expressions and observers" [4].

In my simple app I modularised the download facility which downloads a simple table to PDF. This modularised code can now be reused in a different app or reused within the same app if you for some reason wanted two download facilities!

I created a new R script that contained my modular functions. One function for the UI component and one for the server logic.

```
# Generates the UI download button
downButton <- function(id, label) {
  ns <- NS(id)
  tagList(downloadButton(ns("button")))
}
```

```
# Modular server logic to download a dataframe to a PDF file.
downloadServer <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$button <- downloadHandler(
      filename = "listing.pdf",
      content = function(f) {
        # Create a new empty environment
        # This allows us to pass in only the relevant variables into the report
        e <- new.env()

        e$datasets <- list(nedf)

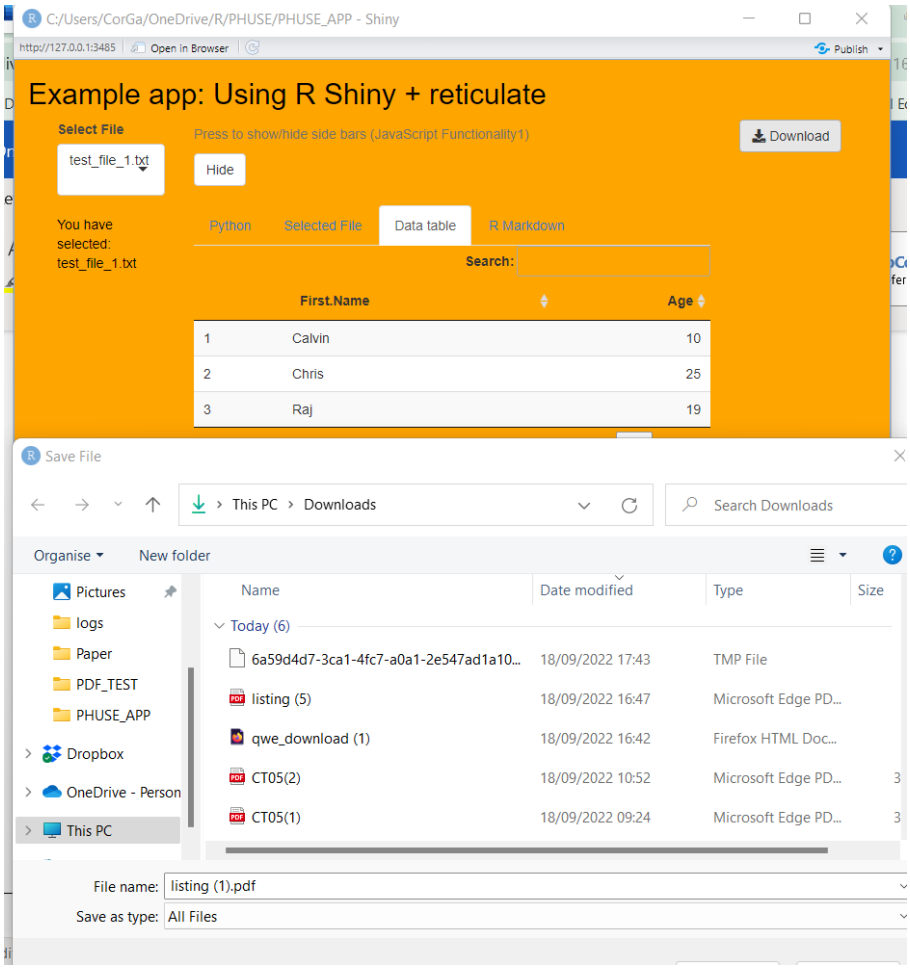
        # Render the document
        rmarkdown::render('template_pdf_v1.Rmd',
                          output_format = rmarkdown::pdf_document(),
                          output_file=f,
                          envir = e)
```

These components are simply called within ui.R and server.R as follows:

```
       # Inserts the UI download button into the app
       downButton("download1", "Download")

       # Inserts the download server logic into the app.
       downloadServer("download1")
```

Once the above code has been set up it enables the downloading of the data table shown in the screenshot below.



R MARKDOWN

As can be seen from the code in the download server R markdown is used to save the table in PDF format. I created a simple R Markdown file which uses the dataframes that form the data table as input and then output these to a PDF file. This functionality is based on great article from Stefan Eng [6].

```
---
title: "Download Example"
author: "Karl Kennedy"
date: "26/5/1999"
output: pdf_document
---

```{r setup, include=FALSE}
library(knitr)
library(pander)
knitr::opts_chunk$set(echo = FALSE)
```

```{r, results='asis'}
panderOptions('knitr.auto.asis', FALSE)
for(d in datasets) {
    pander::pander(d, split.table=120)
}
```
```

The R Markdown code uses two libraries ‘knitr’ which enables the exceution of the R language within the R Markdown file and the pander library to reproduce the table for output.

The 'knitr::opts_chunk$set(echo = FALSE)' line of code prevents the R code executed within the R Markdown file from being displayed in the PDF file.

CONCLUSION

Hopefully this paper along with the supporting articles in References provides a Shiny beginner with enough knowledge to program efficient apps from Day 1. The scope of this paper was to provide a high-level introduction to key Shiny concepts with simple examples. For more in-depth discussion on the concepts please reference the articles below.

Another important concept in Shiny is 'Reactivity' which all Shiny developers need to learn.

REFERENCES

[1] The basic parts of a Shiny app - Shiny Basics
[2] Communicating with Shiny via JavaScript – Joe Cheng
[3] Shiny Reticulate App
[4] Modularizing Shiny app code - Winston Chang
[5] https://emilyriederer.netlify.app/post/shiny-modules/ - Emily Riederer
[6] Dynamic R Markdown Reports with Shiny

ACKNOWLEDGMENTS

I would like to thank Salaja Sirsalewala and John Hirst for their support and encouragement throughout the development of this paper.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Karl Kennedy
GSK
980 Great West Rd, London TW8 9GS
karl.j.kennedy@gsk.com