R Shiny Apps

Requirements

Make sure you have the `Shiny` package installed if you want to follow along this post and try out the examples on your own computer!

It will also be a good idea to get familiar with coding in R if you have not programmed with R before, since your `Shiny` apps will be written in R.

```
# Run this in your console if you don't have the package installed yet
install.packages("shiny")
```

```
# Run this in your console or R Source File to be able to use the package
library(shiny)
```

Introduction

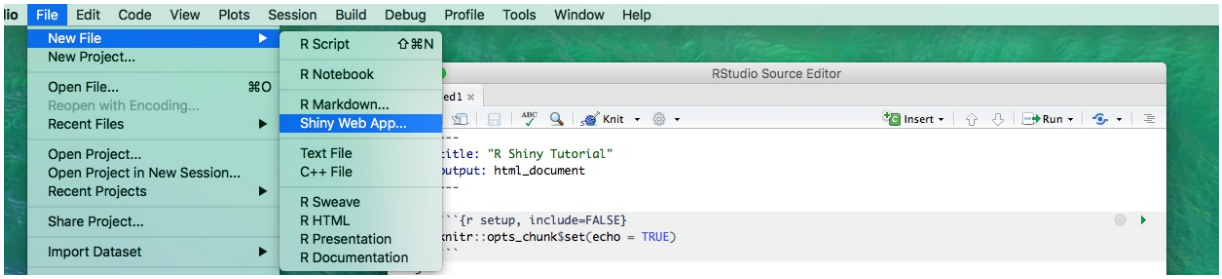
The R `Shiny` package is a web application framework that allows users to deploy interactive web applications wihtout any experience in web development. The complicated tasks of writing HTMl, CSS, and JavaScript are simplified by the `Shiny` package to allow for extremely easy deployment of data visualization web apps.

On a very high level, a `Shiny` app contains two parts: the `UI` which controls the layout and outlook of the web page and the `Server` which controls the logic of your web application. The `Server` monitors the `UI` and dynamically updates the `UI` based on user interactions. If this sounds confusing right now don't worry, we will look over some examples very soon!

This post goes over the basics of how to use `Shiny` and demonstrates some examples of the various functionalities of `Shiny` apps. However, this post by no means covers all you need to know about `Shiny` and R Programming but I hope it will help you get started with creating some beautiful interactive web applications for data visualization.

Some Quick Basics

You can create a new `Shiny` app with a provided basic example and template using `RStudio`.



Shiny_Creation

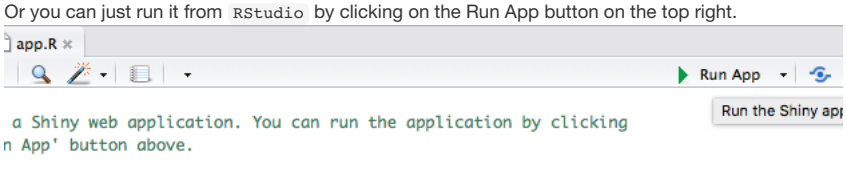
RStudio will prompt you to select a directory to save your application source file.

Note: An R `Shiny` App can be created from separate UI and Server source files or have them all combined into once source file. In this post, we will look through examples using only one source file.

To run your application you can either run the following commands from the R console using absoute or relative paths

```
# Run the following to open up your shiny application
shiny::runApp('Absolute/Relative Path to your shiny application')
```

Or you can just run it from `RStudio` by clicking on the Run App button on the top right.



This example application provides a Histogram of Old Faithful Geyser Data and allows users to dynamically adjust the number of bins in the histogram. Feel free to poke around! We will soon learn how to create our own applications.

This is an example of what a `Shiny` app source file should look like

```
data = ... # Usually some sort of data we desire visualization for is read in and manipulated first.

ui = fluidPage(
  ... #Code that sets up the panels and layout of the web application
)

server = function(input, output) {
  ... #Code that sets up the logic of the application. This UI code will help this section implement dynamic inter
actions with a user.
}

# This actually runs the application
shinyApp(ui=ui, server=server)
```

The UI

Like I mentioned before, `Shiny` apps are made of two parts, the UI and the Server. In this section, we will go over many of the basics in creating the UI. All the code for the UI should go in the section starting with the following:

```
ui <- fluidPage(
  ... #Code that sets up the layout of your web application goes here!
)
```

Note: everything created in the UI section needs to have a variable name to which the server logic can reference and dynamically render outputs based on user inputs.

Most UI use the fluid page which allows a lot of flexibility in creating a desired UI layout. However, you can also use bootstrap pages which is used for working with exisitng prototypes of web applications. There is also a page called `fillPage` which creates a UI page whose height and width will always fill the area of the browset window which is good for scaling content to the size of the window whereas `fluidPage` leaves whitespace at the bottom if the content is smaller than the browser window and has scrolling if the content is larger than the browser window.

Panels

Lets first start with panels. Panel functions are used to group elements of your web application into distinct sections. Every panel in the shiny app will need to have its output specified. Outputs can range from text, to tables, to graphs and many more. A panel can contain multiple other panels, for example, the main panel can contain tab panels to display different charts or graphs based on which tab was selected (this will also require the help of conditional panels). In addition to other panels, each panel will also specify output variables and input widgets.

The output of the panels will be rendered by the server logic using reactive programming. Think of the UI funciton as setting up the skeleton of your web application but for the outputs to actually be seen by the user, the server needs to render them through reactive programming. If this is confusing bear with me, just understand at a high level what panels are and the examples later will help you see how everything comes together.

Here are some of the most important ones you should know, for more information on panels please check out the references section, R documentations and tutorials!:

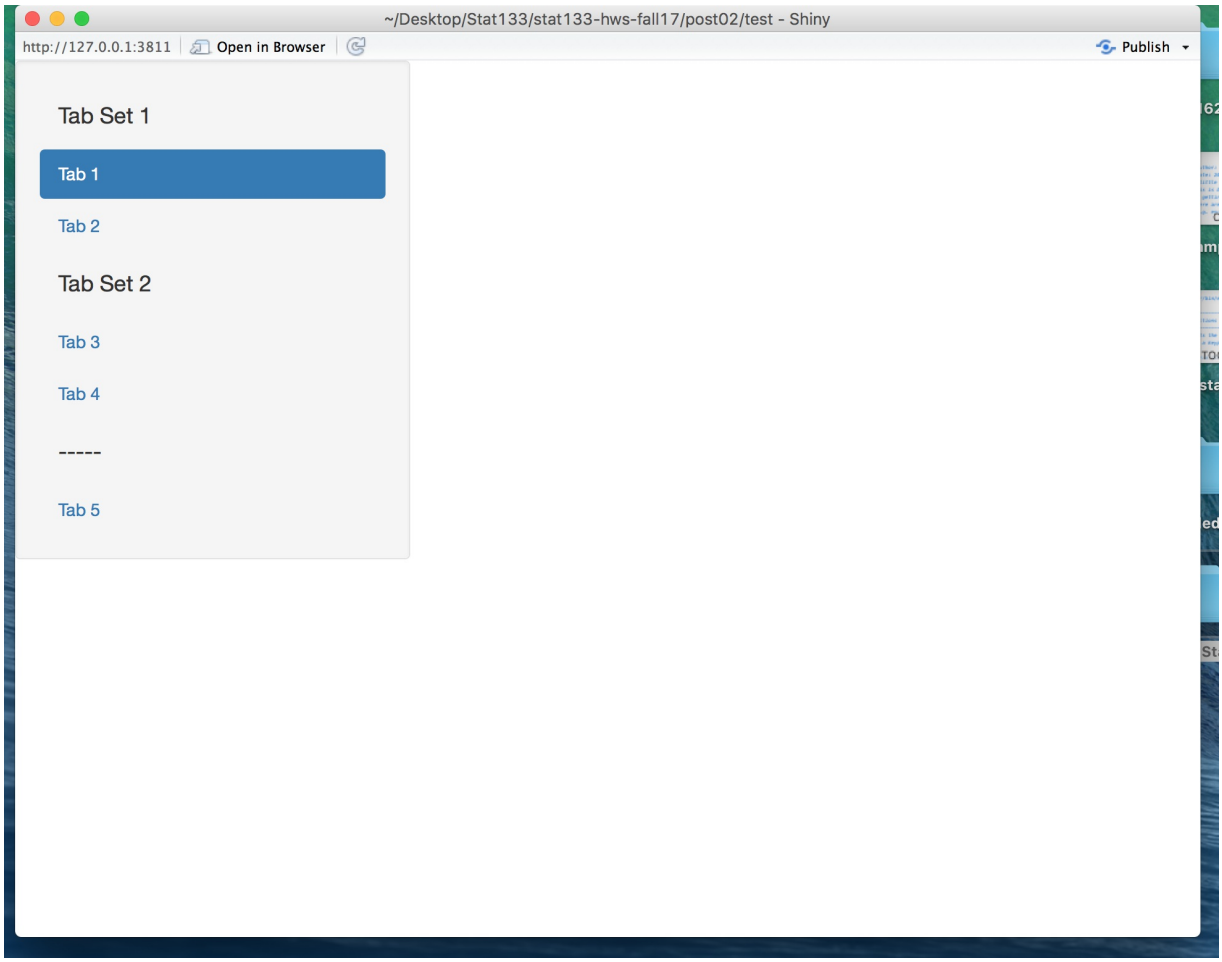
- *titlePanel()*: The title of your web application. This will go on the very top of your application and is always defined first in the code.
- *mainPanel()*: This is the main body of your application.
- *tabPanel()*: This creates a tab panel. This panel works closesly with conditional panels to allow for users to see different sidebar panels or other panels with different tabs of the web application. This logic will be demonstrated in an example below. Each tab panel can be given a value in whcih a conditional panel will reference to see if this tab should be rendered.
- *conditionalPanel()*: A conditional panel is just like any other panel but will only show itself on the web application if a condition is met.
- *sidebarPanel()*: A panel that goes on the side like a side bar. These are usually used to contain user input widgets to help create a very clean user interface of the web application.
- *navlistPanel()*: Presents various components as a sidebar list instead of using tabs.

This is an example code snippet of using some panel functions:

```
# This creates a mainPanel that displays a text and 1 tab panel that displays a plot.
ui <- fluidPage(
  mainPanel(
    tabsetPanel(type= 'tabs',
                tabPanel("TestTab1",
                         plotOutput("TestPlot"))
    ),
    textOutput("Test")
  )
)
```

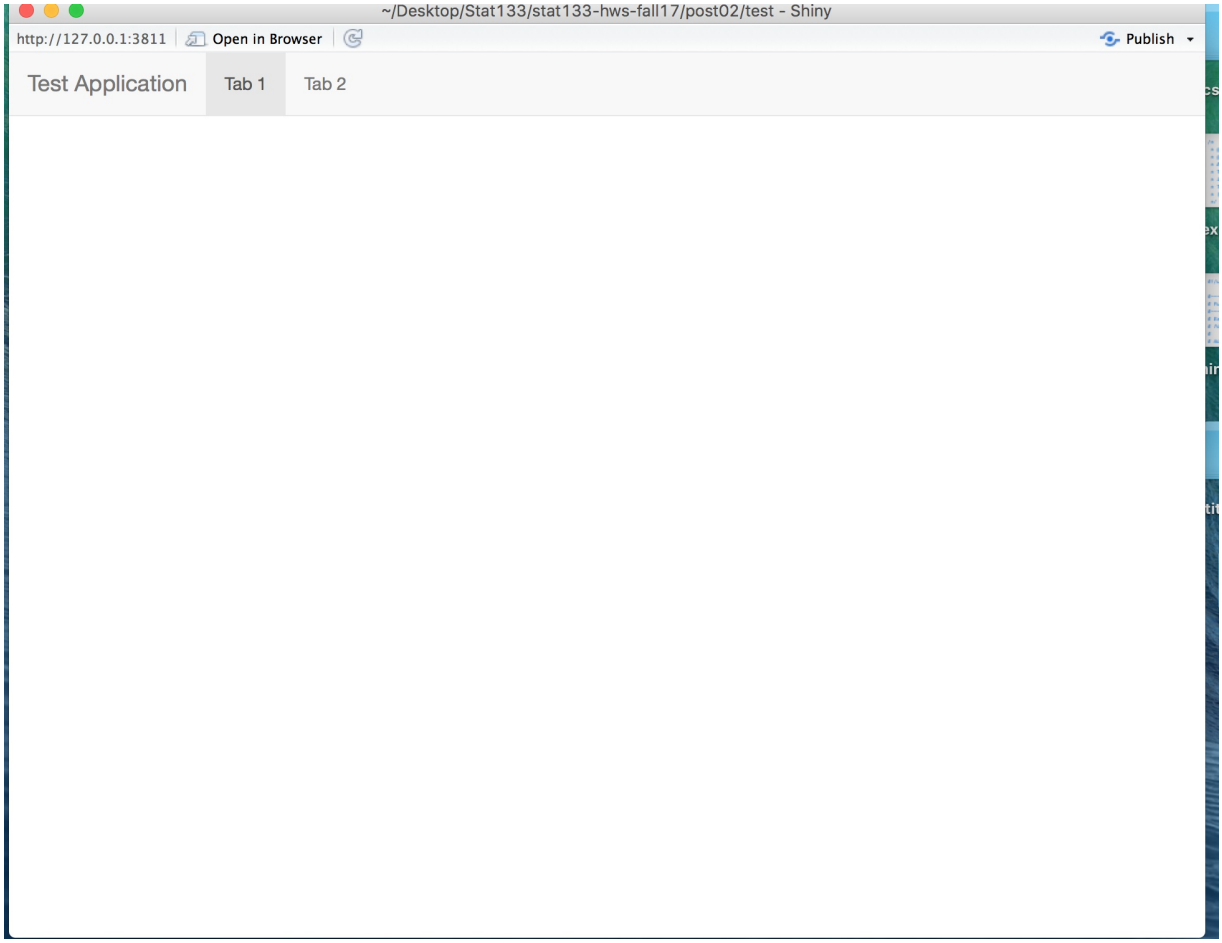
Tab Panels are great for separating your app into sections but navlist is a good panel to use if you have a lot of tabs. It congregates all your tabs into a sidebar list. Or you can even make a navbarPage which creates a UI page that has a Navbar bootstrapped at the top for a very clean look. Here are some code examples

```
#Code for using a navlist panel to list sections of your work instead of tabs.
ui <- fluidPage(

  navlistPanel(
    "Tab Set 1",
    tabPanel("Tab 1"),
    tabPanel("Tab 2"),
    "Tab Set 2",
    tabPanel("Tab 3"),
    tabPanel("Tab 4"),
    "-----",
    tabPanel("Tab 5")
  )
)
```



Notice how the majority of the page is blank, this is because we didn't specify any outputs for our panels yet.

```
# A UI Page with a navbar boostrapped to the top.
ui <- navbarPage(
  "Test Application",
  tabPanel("Tab 1"),
  tabPanel("Tab 2")
)
```



A Navbar Panel Looks like this

Again there is nothing on the page because we haven't written any outputs yet.

Layouts

Layout functions are used to organize panels and elements into an exisitng layout in the package. Layouts specify how your application is going to look. Without a layout, the panels and elements we create will look sparse and not as nicely organized on the web application.

Here are some important layouts, again for more information please checkout the references section, R documentations and tutorials.

- *flowLayout()*: This layout puts your panels and elements in a left-to-right, top-to-bottom arrangement
- *sidebarLayout()*: This is probably the most used layout. This puts a sidebar panel on the left and a main panel on its right.
- *splitLayout()*: This layout puts elements into sections horizontally, dividing the space evenly.

This is an example code snippet of how to use the sidebar layouts function

```
ui <- fluidPage(
  sidebarLayout(
        sidebarPanel(
           ... #Code for a sidebarPanel
        ),

        mainPanel(
           ... #Code for the mainPanel of the sidebarLayout
        )
    )
)
```

Input Widgets

Input widgets are built in gadgets that allow various forms of user input on the web application. Input widgets are great for dynamic user interactions with the `Shiny` app and varying visualizations of the data plots. The way dyanmic interaction works with input widgets is that a user chooses a selection for an input widget on the website UI, then the server logic and read in the widget selection variable and determine what to do with it, whether it be rendering a new text or changing the data plot.

Input widgets are created inside a panel function of the UI but the actual effects of the user inputs such as specifying smaller bin widths for a histogram have to be rendered in the server function logic. The input choices for the widgets will be specified in the UI as well.

Input widgets are created with built in functions of the `Shiny` package and requires a varaible name, a title name, selection choices, and other arguments.

Here are some input widgets, check the references section for examples of all the widgets.

- *actionButton*: An action button.
- *checkboxInput*: A checkbox for some input.

- *radioButtons*: A set of radioButtons that provide different options.
- *selectInput*: A box with choices ot select from. The selections need to be passed in as an argument.
- *sliderInput*: A slider bar with various values you can set the bar on the slider at.

Here is an example code snippet of how to use a input widget

```
# This creates a slider input widget, titled Number of Bins on the side bar panel. It has a minimum value of 1, a maximum of 50 and a default value of 30. The string 'bins' is the variable name of this slider input which will be used to implement the server logic.
sidebarPanel(
  sliderInput("bins", "Number of Bins", min=1, max=50, value=30)
)
```

Output

As mentioned earlier, each panel needs to specify its outputs. These outputs are what will get rendered in the server logic. There are many types of outputs and here are some you should become familiar with:

- *TextOutput*
- *ImageOutput*
- *PlotOutput*
- *TableOutput*

There can also be special outputs, for example, when using the `ggvis` library to make plots, you can specify a `ggvisOutput` for a panel.

These outputs need to be specifically referenced for rendering in the Server code, we will see an example soon.

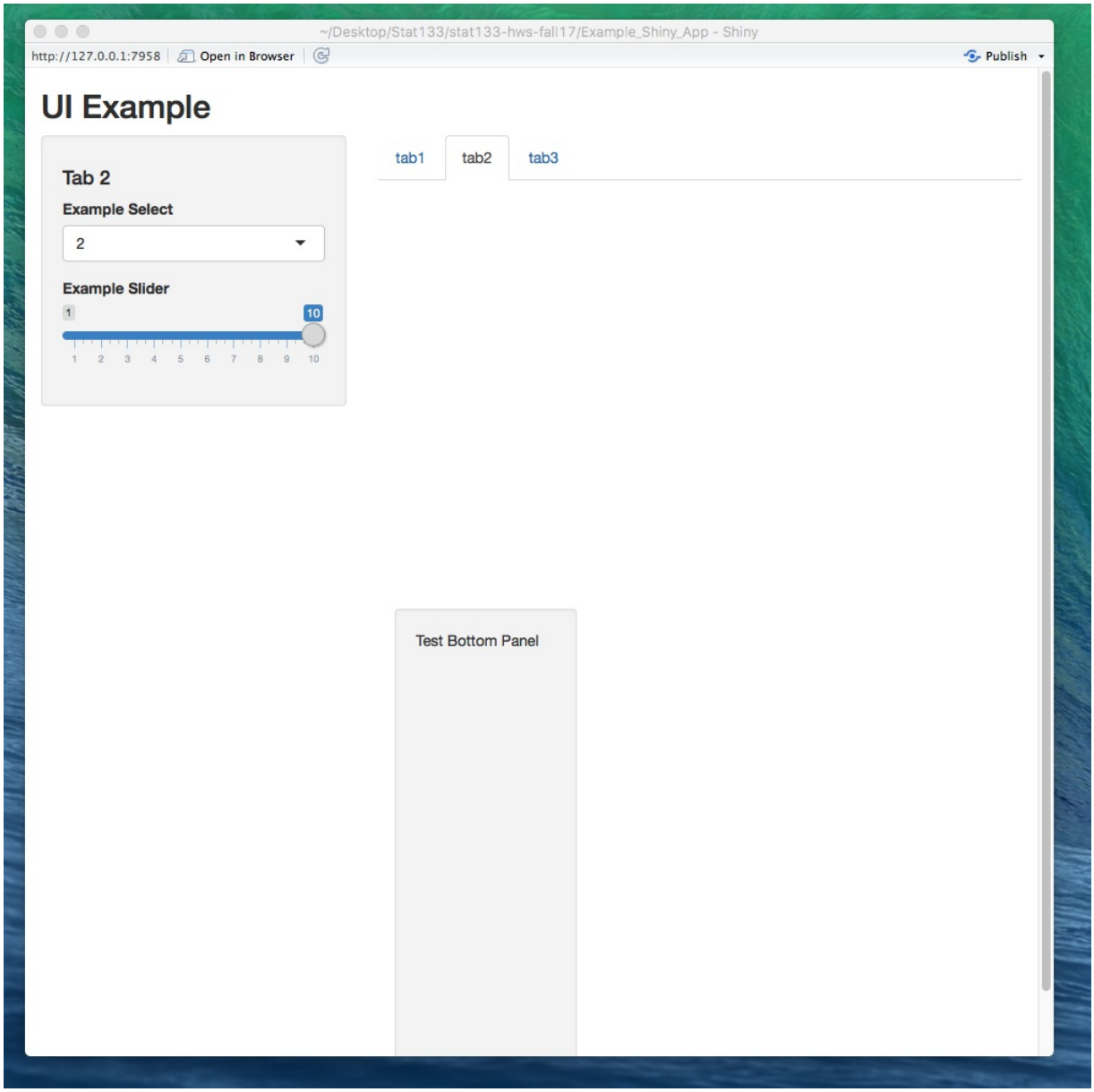
UI example

Lets put everything we learned together about the UI so far and go through an example!

Put the following code in your Shiny App Source Code File and run the app

```
ui <- fluidPage(

  # Application title
  titlePanel("UI Example"),

  # We will be using a sidebarLayout for our web app.
  sidebarLayout(
    sidebarPanel(
      # The first tab in the sidebar. It will only be rendered if the tabselect == 1.
      conditionalPanel(condition = "input.tabselected==1",
                       h4("Tab 1") # The header of this tab panel.
                       ),
      # The second tab in the sidebar panel. Contains a select input widget and a slider input widget.
      conditionalPanel(condition= "input.tabselected==2",
                       h4("Tab 2"),
                       selectInput("var2", "Example Select", c(1, 2, 3),
                                   selected = 2),
                       sliderInput("slide", "Example Slider",
                                   min=1, max=10, value=10)
                       ),
      # The third tab in the sidebar panel. Contains a radio button input widget.
      conditionalPanel(condition= "input.tabselected==3",
                       radioButtons("radio", "Example Radio", choices=list(
                         "none" = 1, "choice1" = 2, "choice2" = 3), selected = 1)
                      )
    ),

    # The main panel of our web app layout. Our main panel will be divided into three tabs which the user can sele
ct. Each tab has a value which is referenced by the conditionalPanels of the sidebarPanel to change the sideBar pa
nel based on tab selection.
    mainPanel(
      tabsetPanel(type = "tabs",
        tabPanel("tab1", value=1,
                 textOutput("sample_output1")
        ),
        tabPanel("tab2", value=2,
                 plotOutput("sample_output2"),
                 sidebarBottomPanel("Test Bottom Panel", plotOutput("sample_output3"))
        ),
        tabPanel("tab3", value=3,
                 plotOutput("sample_output4")
        ),
        id = "tabselected"
      )
    )
  )
)

# Define server logic required to draw a histogram
server <- function(input, output) {
  # We will ignore server logic for now.
}

# Run the application
shinyApp(ui = ui, server = server)
```

As mentioned before, keep in mind that we won't actually see any of the outputs we defined in the UI code because we haven't implemented server logic to render them yet. The code above will only show the overall layout of the UI, it should look something like this:



Example UI

Your WebApp UI should be a sidebarLayout with three tabs in the main panel you can choose from and different sidebar panels with different input widgets depending on which tab you choose.

Fixed Page vs Fluid Page

The UI can be implemented as a `fixedPage` or a `fluidPage`. Both layouts use a flexibly sub-dividable 12-column grid which you divide up into sections for your content.

The examples above all deal with a fluidPage implementation of the UI. `fluidPages` allows use of high level layout functions such as `sidebarLayout` and makes it a lot easier in creating a UI layout for organizing your panels of content. The `fluidPage` will dynamically resize your content as the size of the page changes.

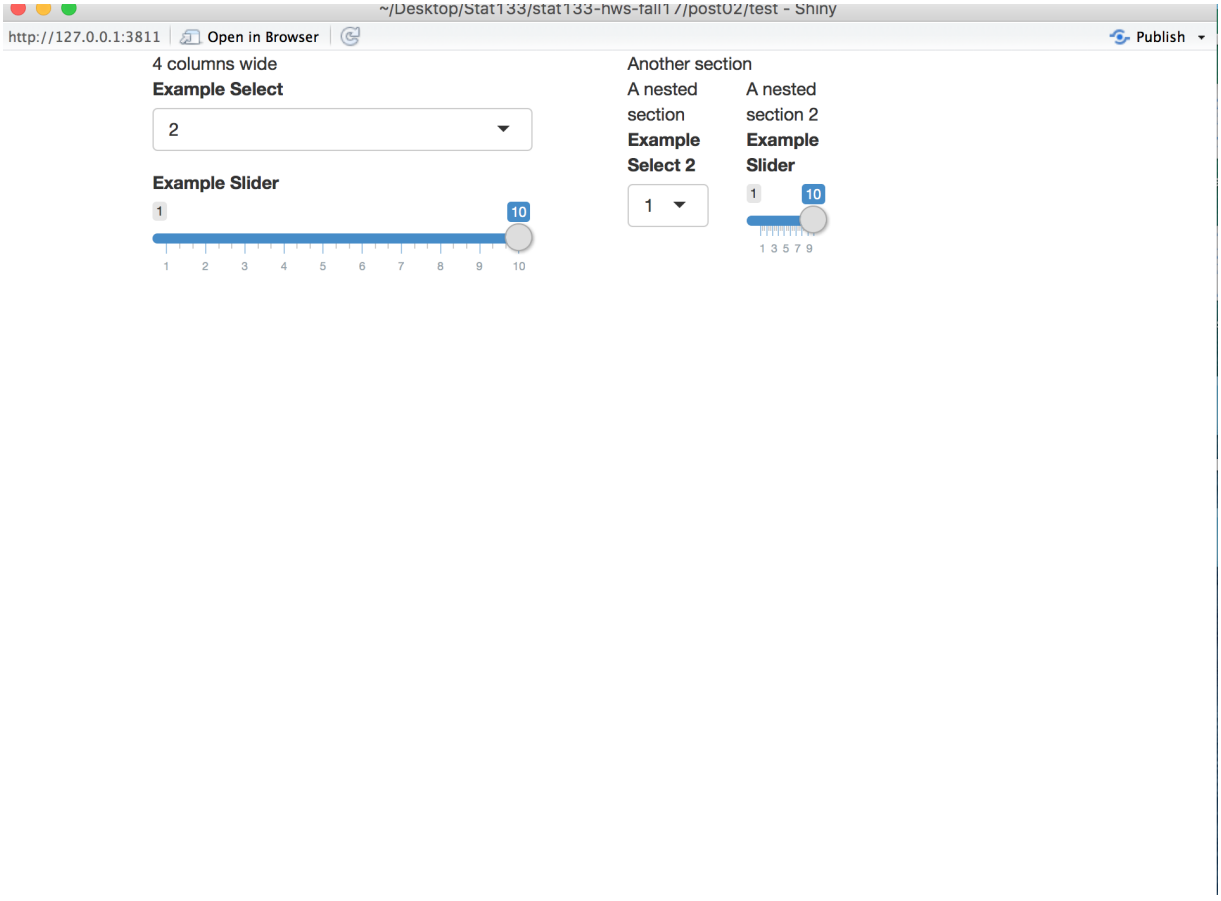
A `fixedPage` requires you to use the `fixedRow()` function to build your UI layout yourself and can't use the higher level layout functions. You have to specify exactly where to put your content. Within the `fixedRow()` funciton you have to use the `column()` function to create sections of content. The `column()` function allows you to create section of a desired size. The `width` parameter allows you to specify how many columns wide this section should be. There is also an optional `offset` parameter in the `column` function which is used on the center input column to provide custom spacing between the first and second columns. Finally you can use panel functions and input widgets inside the `column` functin to create desired panels and input UI.

Additionally you can nest these fixed rows and columns. For example, say you divided your page into two sections of 6 columns wide each. You can create more sections under each of these two sections with `fixedRow()` and the `column()` function again.

Here is an example of a `fixedPage` to help better understand how it works.

```
# This creates a fixedPage UI layout with a row of content that is divided into two sections, and one of the secti
ons will have two subsections
ui <- fixedPage(
  titlePanel="FIXED PAGE",

  fixedRow(
    column(width = 6,
           "4 columns wide",
           selectInput("var", "Example Select", c(1, 2, 3),
                       selected = 2),
           sliderInput("slide", "Example Slider",
                       min=1, max=10, value=10)
    ),
    column(width = 6,
           "Another section",
           fixedRow(
             column(width = 3,
                    "A nested section",
                    selectInput("var", "Example Select 2", c(1,2,3))),
             column(width = 3,
                    "A nested section 2",
                    sliderInput("slide", "Example Slider",
                       min=1, max=10, value=10)
                    )
           )
    )
  )
)
```

The UI layout created by this code will look like this:



A Fixed Page Example

This might be a confusing topic so make sure you check out the application layout guide linked in the references to gain a better understanding!

The Server

Let's first reinforce our knowledge of how a `Shiny` 's server function interacts with its UI. We have mentioned before that each of the UI elements will need a variable name, and that the variable names are what the server logic to reference to allow dynamic user interaction with the web app and rendering of different plots and such based on different user inputs. This is what we call Reactive Programming. Reactive programming is what allows our outputs to update automatically based on input changes.

Reactive programming is a model of programming used by `Shiny` apps in which a reactive source, reactive conductor, and reactive endpoint is involved.

The source is user input through the UI that we built. For example, moving the slider bar on the sliderInput widget to a certain value or selecting a certain option from a sectionInput will set the source values of the variable name associated with these input widgets. These input values can then be accessed in the server function using this syntax:

```
server = function(input, output) {
  # This retrieves the whatever value the user chose sample_input_variable to be through its widget in the website
UI.
  input1 = input$sample_input_variable
  ...
}
```

You don't have to worry too much about reactive conductors for basic shiny apps. They are usually some sort of function that can help encapsulate slow operations and speed up calculations of desired output values. You can read more about them online if you are interested in how they work!

A reactive endpoint is our output elements for the web page such as graphs, texts, tables, and others. They can be told in the server logic to re-execute and re-render to reflect changes of the input values. For example, if a user decides to select another variable of a dataset to plot through the selectInput widget on a line graph, the server logic will re-render the plot to the appropriate variable.

Note: `Shiny` will automatically make an object reactive if it uses an input value.

Explicit output variables such as textOutputs can be rendered with render functions using the following syntax:

```
server = function(input, output) {
  # This will display out text output as whatever the user set the sample_input_variable to.
  output$output_text_variable_name = renderText(input$sample_input_variable)
  ...
}
```

You can create reactive graphs using libraries such as ggvis in the following way:

```
server = function(input, output) {
    # The following code creates a reactive histogram that is rendered for the specified user input every time the
user input changes for some variable, probably through some selectInput widget. Remember every widget will have a
variable name that the server logic can reference.
    vis_histogram = reactive({
    var = input$variable

    some_data_frame_to_be_plotted %>%
      ggvis(x = var) %>%
      layer_histograms(stroke := 'white',
                       width = input$width)
  })

  vis_histogram %>% bind_shiny("histogram")
}
```

Note that you can create graph outputs in many other ways in R such as with ggplot and they can all be rendered by shiny, just make sure you use the correct libraries.

Remember that these ouput and input variables are first created in the UI, for example, the outputs specified in the panels we set up. The server function is in charge of bringing these variables and elements to life and rendering them on the web page for the user of the application.

Here is a list of some render functions for outputs:

- *renderImage*: This function is in charge of creating images for imageOutputs you specified for panels in the UI (saved as a link to a source file).
- *renderPlot*: This function renders plots for plotOutputs you specified in the UI.
- *renderTable*: This function is used to render tableOutputs you specified in the UI.

Putting it All Together

Let's now put everything together and try to create an example `Shiny` web application of our own.

For this example, we will be working with NBA player data from the 2016-2017 season.

Before we start, make sure you have the csv data downloaded and stored in the directory with your app's source file.

You can find the data's csv file and its description file on this GitHub link:

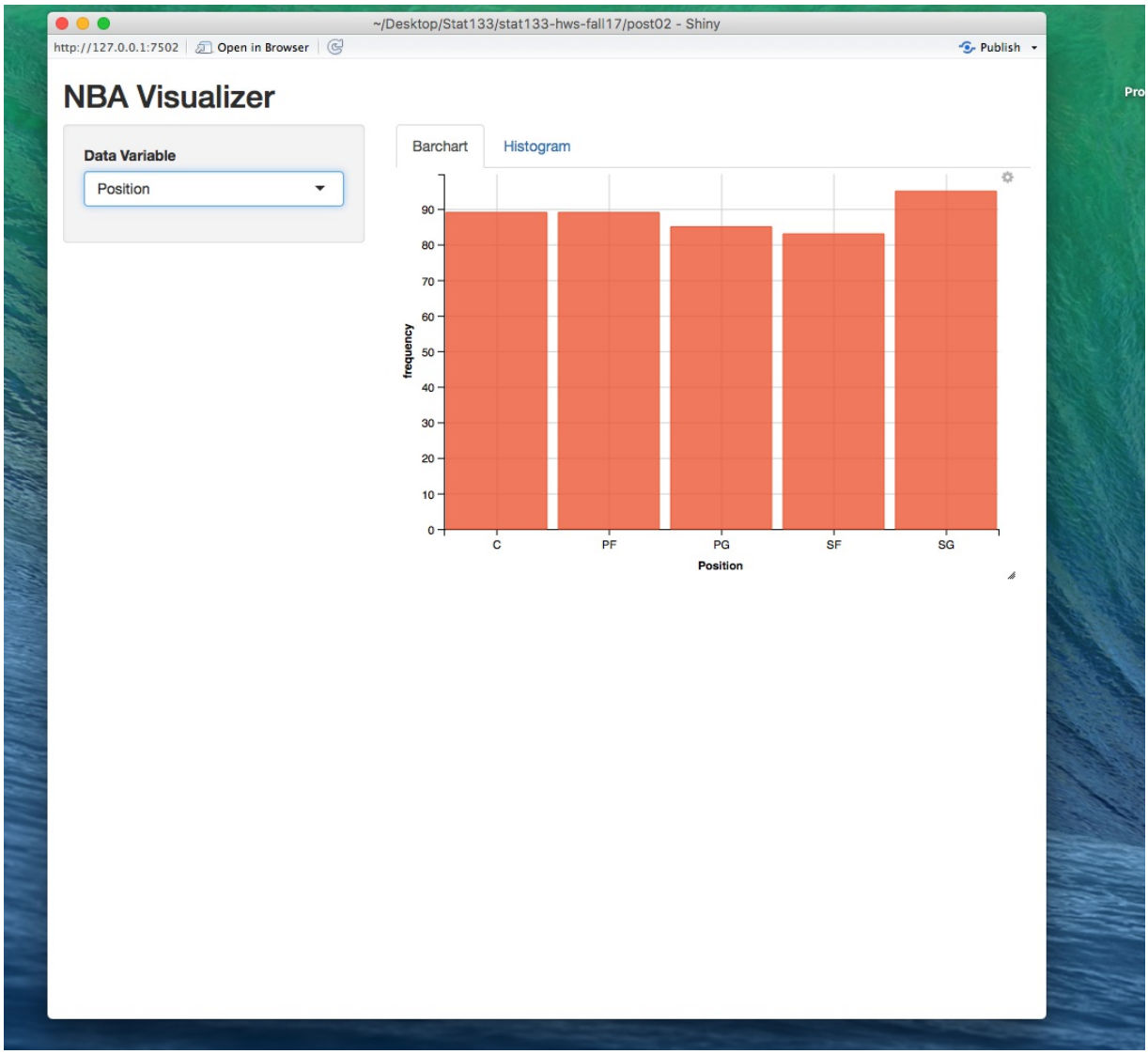
https://github.com/cody-zhang/stat133-hws-fall17/tree/master/post02/data

To download the data, you can click on the csv file, click on Raw on the top right and then click Save as…

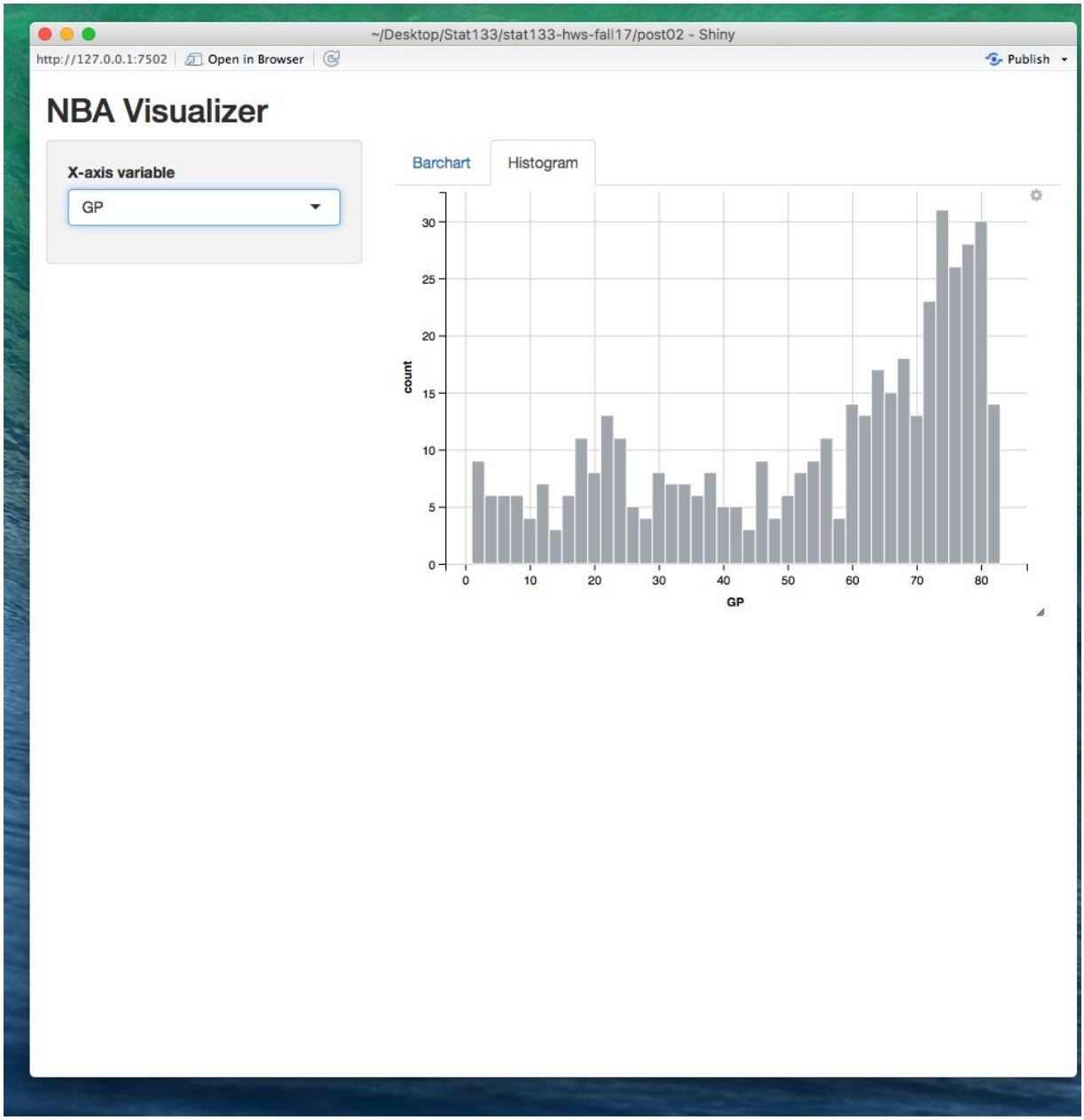
Put the following code into your `Shiny` app's source file

```
# First load the libraries that we will need to use
library(shiny)
library(ggvis)

# We then read in the data from the csv file saved in whichever directory. Remember to use relative paths.
nba_data = read.csv('./data/nba2017-players-statistics.csv', stringsAsFactors = FALSE)

# Separate the data into its quantitative and categorical variables.
categorical = names(nba_data)[c(2:4)]
quantitative = names(nba_data)[c(5:24)]

# Define the UI of the application
ui <- fluidPage(

  # Application title
  titlePanel("NBA Visualizer"),

  # We will be using the sidebarLayout with the sidebarPanel using conditional panels.
  sidebarLayout(
    sidebarPanel(
      conditionalPanel(condition = "input.tabselected==1",
                       textOutput("text1"),
                       selectInput("var1", "Data Variable", categorical,
                                   selected = "Team")
      ),
      conditionalPanel(condition = "input.tabselected==2",
                       textOutput("text2"),
                       selectInput("var2", "X-axis variable", quantitative,
                                   selected = "GP")
      )
    ),

    # The main panel will have our tabs and graphs which will be generated by using ggvis
    mainPanel(
      tabsetPanel(type = "tabs",
                  tabPanel("Barchart", value=1,
                           ggvisOutput("barchart")
                  ),
                  tabPanel("Histogram", value=2,
                           ggvisOutput("histogram")
                  ),
                  id = "tabselected"
      )
    )
  )
)

# Define server logic
server <- function(input, output) {
  # The graphs are drawn using ggvis, type ?ggvis in your console to see what the different arguments mean!

  # Our graphs need to be made reactive. Note how we are using the input variable name to get user inputs?
  vis_barchart = reactive({
    # The following code converts the input variable into something we can use in ggvis.
    var1 = prop("x", as.symbol(input$var1))

    # We render the graph based on the user input.
    nba_data %>%
      ggvis(x = var1, fill := "#ef623b") %>%
      layer_bars(stroke := '#ef623b', fillOpacity := 0.8, fillOpacity.hover := 1) %>%
      add_axis("y", title = "frequency")
  })

  vis_barchart %>% bind_shiny("barchart")

  output$text1 = renderText ({"Bar Graphs for NBA Team, Position, And Experience"})

  # Make the histogarm the exact same way
  vis_histogram = reactive({
    var2 = prop("x", as.symbol(input$var2))

    nba_data %>%
      ggvis(x = var2, fill := "#abafb5") %>%
      layer_histograms(stroke := 'white',
                       width = input$width)
  })

  vis_histogram %>% bind_shiny("histogram")

  output$text2 = renderText({"Histograms for NBA Data"})
}

# Run the application
shinyApp(ui = ui, server = server)
```

In this example, we created a web application that visualizes the NBA data with a sidebar layout. There are two different tabs, one for visualization of categorical variables using bar charts in the data and one tab for visualization of quantitative data using a histogram. Each tab will also have its own sidebar with input widgets that allow you to select different variables to visualize. Run the source file and you should see a webpage pop up looking like the following.



Bar Charts Tab



Histograms Tab

Note how there are two tabs you can select from in the main panel of your application and that each tab has its own side bar panels. You can select different variables to visualize using the input widgets and see the graphs change!

Congratualations you just made your first basic `Shiny` Application!

Work Flow Summary

In summary the work flow of how a `Shiny` web application works is as follows:

1. We create a UI with a specific layout and panels. This is where we decide how our UI will look and how to divide up our content into sections.
2. We add input and output elements to our UI (i.e. input widgets, plotOutputs, etc). These each have a variable name that can be referenced. The user is able to change the input values through interacting with the UI.
3. In our server logic, we will listen for the input values of our input elements. Based on the user inputs, we can take appropriate action to render the output using specific render functions. For example, if the user specifies a certain bin width value on a slider input widget, we can dynamically render a new histogram on the web page of the specified bin widths.Remember that objects are automatically reactive if it uses an input variable but you can also specify for things to be reactive.

Note: We must be very very careful with creating and referencing variable names of our elements so that the web page functions desirably.

References

Here are some very good references for you to explore using `Shiny` further. Hopefully they can help you build more and more beautiful web applications!

A Great Beginner Tutorial To Shiny

A Guide On Application Layouts for Shiny

A Great Overview of Reactivity in Shiny Apps

References to All the Shiny Functions

A Gallery of All the Input Widgets and How to Use them, Very Helpful!

A Site Showcasing Various Wonderful Apps Built With Shiny

An Interactive Tutorial To Help You Build A Shiny App

A Post Showing Various Packages That Can Make Shiny Apps Even Better!

Take Home

Hopefully this post helped you become more familiar with using the `Shiny` R package. Data is everywhere in today's world and is continuously growing. Shiny Web Apps are a great way for you to visualize these data sets and share it with users, helping analyze the data at hand and finding important trends that can be very useful to understanding the world.