Data visualization with googleVis exercises part 7

Table, Org Chart & Tree Map

In the seventh part of our series we are going to learn about the features of some interesting types of charts. More specifically we will talk about Table, Org Chart and Tree Map.

Read the examples below to understand the logic of what we are going to do and then test yous skills with the exercise set we prepared for you. Lets begin!

Answers to the exercises are available here.

Package

As you already know, the first thing you have to do is install and load the googleVis package with:

```
install.packages("googleVis")
library(googleVis)
```

NOTE: The charts are created locally by your browser. In case they are not displayed at once press F5 to reload the page.

Table

It is quite simple to create a Table with googleVis. We will use the “Stock” dataset.
Look at the example below to create a simple table:

```
TableC <- gvisTable(Stock)
plot(TableC)
```

Exercise 1

Create a list named “TableC” and pass to it the “Stock” dataset as a table. **HINT**: Use gvisTable().

Exercise 2

Plot the the table. **HINT**: Use plot().

Table with pages

To add pages to your table use:
options=list(page='enable')

Exercise 3

Add pages to the table you just created and plot it. **HINT**: Use list().

Org chart

It is quite simple to create an Org Chart with googleVis. We will use the “Regions” dataset. You can see the variables of your dataset with head().
Look at the example below to create a simple Org Chart:

```
OrgC <- gvisOrgChart(Regions )
plot(OrgC)
```

Learn more about using GoogleVis in the online course Mastering in Visualization with R programming. In this course you will learn how to:

- Work extensively with the GoogleVis package and its functionality
- Learn what visualizations exist for your specific use case
- And much more

Exercise 4

Create a list named “OrgC” and pass to it the “Regions” dataset as an org chart. **HINT**: Use gvisOrgChart().

Exercise 5

Plot the the org chart. **HINT**: Use plot().

Dimensions

You can adjust the dimensions of the org chart with these options:

```
options=list(width=600, height=250,
size='large')
```

Exercise 6

Adjust the dimensions of your org chart. Set height to 300, width to 550 and size to medium and plot it.

Tree Map

It is quite simple to create a Tree Map with googleVis. We will use the “Regions” dataset.
Look at the example below to create a simple Tree Map:

```
TreeC <- gvisTreeMap(Regions)
plot(TreeC)
```

Exercise 7

Create a list named “TreeC” and pass to it the “Regions” dataset as an org chart. **HINT**: Use gvisTreeMap().

Exercise 8

Plot the the tree map. **HINT**: Use plot().

You can decide tha dependents variables of your dataset by selecting it. In the example above the dependent variable was “Val”. To choose “Fac” follow the example:

```
TreeC <- gvisTreeMap(Regions,
"Region", "Parent",
```

"Fac")
plot(TreeC)

Exercise 9

Set “Fac” as your dependent variable, plot the tree map and see the difference.

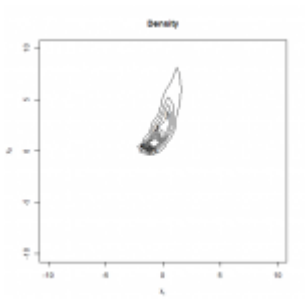
Font size

Obviously you can change the font size of your tree map simply with:
options=list(fontSize=10)

Exercise 10

Set the size of your font to 20 and plot your tree map. **HINT**: Use fontSize.

Shiny Application Layouts Exercises (Part-10)



Shiny Application Layouts – Shiny Themes

In the last part of the series we will check out which themes are available in the shinythemes package. More specifically we will create a demo app with a selector from which you can choose the theme you want.

This part can be useful for you in two ways.

First of all, you can see different ways to enhance the appearance and the utility of your shiny app.

Secondly you can make a revision on what you learnt in "Building Shiny App" series and in this series, as we will build basic shiny staff in order to present it in the proper way.

Read the examples below to understand the logic of what we are going to do and then test yous skills with the exercise set we prepared for you. Lets begin!

Answers to the exercises are available here.

Create the App.

In order to see how themes affect the application components ,we have seen until now, we need to create them.

As we are going to use tags here, a good idea is to use taglList() in order to create our app. TagList() is ideal for users who wish to create their own sets of tags.

Let's see an example of the skeleton of our application and then create our own step by step before applying to it the theme selector.

```
#ui.R
tagList(
navbarPage(
"Title",
tabPanel("Navbar 1",
sidebarPanel(
sliderInput("slider","Slider input:", 1, 100, 50),
tags$h5("ActionButton:"),
actionButton("action", "Action")
),
mainPanel(
```

```
tabsetPanel(
tabPanel("Table",
h4("Table"),
tableOutput("table")),
tabPanel("VText",
h4("Verbatim Text"),
verbatimTextOutput("vtxt")),
tabPanel("Header",
h1("Header 1"),
h2("Header 2"))
)
)
),
tabPanel("Navbar 2")
))
#server.R
function(input, output) {
output$vtxt <- renderText({
paste(input$slider)
})
output$table <- renderTable({
iris

})
}
```

Exercise 1

Create a UI using tag List with the form of a Navbar Page and name it “Themes”. **HINT**: Use tagList() and navbarPage().

Exercise 2

Your Navbar Page should have two tab Panels named “Navbar 1” and “Navbar 2”. **HINT**: Use tabPanel().

Exercise 3

In “Navbar 1” add sidebar and main panel. **HINT**: Use

sidebarPanel() and mainPanel().

Exercise 4

Create three tab panels inside the main panel. Name them “Table”, “Text” and “Header” respectively. **HINT**: Use tabsetPanel() and tabPanel().

Exercise 5

In the tab panel “Table” add a table of the iris dataset. Name it “Iris”. **HINT** : Use tableOutput() and renderTable({}).

Exercise 6

In the tab panel “Text” add verbatim Text nad name it “Vtext”. **HINT**: Use verbatimTextOutput().

Exercise 7

Add a slider and an actionbutton in the sidebar. Connect the slider with the “Text” tab panel. Use tags to name the actionbutton. **HINT**: Use sliderInput(),actionButton(), renderText() and tags.

Exercise 8

In the tab panel “Header” add two headers with size h1 and h2 respectively.

Shinythemes

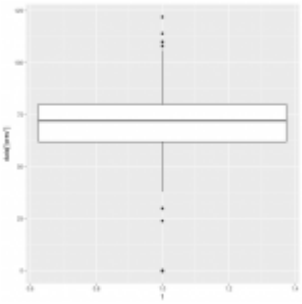
Exercise 9

Install and load the package shinythemes.

Exercise 10

Place themeSelector() inside your tagList() and then use different shiny themes to see how they affect all the components you created so far.

Shiny Application Layouts Exercises (Part-8)



Shiny Application Layouts – Dynamic UI

In the eightth part of our series we will see how we can build a user interface with dynamically generated components.

The UI components are generated on the server side with the use of renderUI() and are displayed with uiOutput() on the ui side. Every time a new command is sent by the user, it replaces the previous command.

This part can be useful for you in two ways.

First of all, you can see different ways to enhance the appearance and the utility of your shiny app.

Secondly you can make a revision on what you learnt in "Building Shiny App" series as we will build basic shiny staff in order to present it in the proper way.

Read the examples below to understand the logic of what we are

going to do and then test yous skills with the exercise set we prepared for you. Lets begin!

Answers to the exercises are available here.

Application Context

First of all let's build the skeleton of our app like the example below.

```
#ui.R
fluidPage(
titlePanel("Title"),
fluidRow(

column(4,wellPanel()),
column(4,wellPanel()),
column(4,wellPanel())))
#server.R
function(input, output) {}
```

Exercise 1

Create the initial fluid Page with a title. **HINT**: Use fluidPage() and titlePanel().

Exercise 2

Create a row, separate it with columns of size = 3 and add a well Panel in each one of them. **HINT**: Use fluidRow(), column() and wellPanel().

Below is an example of a selectInput() from which the user is going to choose the dynamic component he wants.

```
#ui.r
selectInput("input_type", "Input type",
c("slider", "text", "numeric", "checkbox",
"checkboxGroup", "radioButtons", "selectInput",
"selectInput (multi)", "date", "daterange"
)
)
```

Exercise 3

Add a selectInput() in the first well Panel with: slider input, text input, numeric input, check box, radio buttons and date.

The uiOutput() generates the dynamic UI component.

```
#ui.R
uiOutput("ui")
```

Exercise 4

In the second well Panel add the function that is going to generate your ui output.

In the third well Panel we are going to create two “boxes”. The first one will display the type of the input that the user chooses every time while the second the value of this input. Both of them will be displayed as text. So we have to use verbatimTextOutput() and use tags for the titles. Look at the ui side example below:

```
#ui.R
tags$p("Title 1:"),
verbatimTextOutput("input_title_1"),
tags$p("Title 2:"),
verbatimTextOutput("input_title_2")
```


Exercise 5

In the third well panel should automatically be typed the type of the input of your choice as long as the value of this input. Create only the ui side. **HINT**: Use verbatimTextOutput().

Exercise 6

Name the two components that you just created. **HINT**: Use tags.

Reactivity

Now let’s go to the server side to make things reactive like this:

```
#server.R
output$ui <- renderUI({})
```

Exercise 7

Put your app in a reactive context. **HINT**: Use renderUI().

Now we have to connect the dynamic components we created in **Exercise 3** with their server side. Below is an example:

```
#server.R
"slider" = sliderInput("dynamic", "Dynamic",
min = 1, max = 100, value = 50),
"text" = textInput("dynamic", "Dynamic",
value = "EXAMPLE"),
"numeric" = numericInput("dynamic", "Dynamic",
value = 10),
"checkbox" = checkboxInput("dynamic", "Dynamic",
value = TRUE),
"radioButtons" = radioButtons("dynamic", "Dynamic",
choices = c("Option 1" = "option1",
"Option 2" = "option2"),
selected = "option1"
),
"date" = dateInput("dynamic", "Dynamic"))
```

Exercise 8

Create the server side of the dynamic components you created in **Exercise 3**. Put values of your choise but make sure they are connected with the ui side.

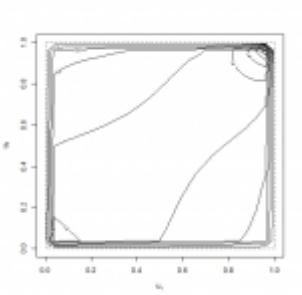
Exercise 9

Display the name of the dynamic component of your choice. **HINT**: Use renderText().

Exercise 10

Display the value of the dynamic component that you chose. **HINT**: Use renderPrint().

Shiny Application Layouts Exercises (Part-7)



Shiny Application Layouts – Conditional Panel

In the seventh part of our series we will use the rnorm() function to create a UI with a Conditional Panel.

This type of Panel is visible only when the value of a JavaScript expression is true. The JS expression is re-evaluated every time shiny runs with a different input.

This part can be useful for you in two ways.

First of all, you can see different ways to enhance the appearance and the utility of your shiny app.

Secondly you can make a revision on what you learnt in "Building Shiny App" series as we will build basic shiny staff in order to present it in the proper way.

Read the examples below to understand the logic of what we are going to do and then test yous skills with the exercise set we prepared for you. Lets begin!

Answers to the exercises are available here.

Exercise 1

Create the initial fluid page. **HINT**: Use fluidPage().

Exercise 2

Name your application with a title. HINT: Use titlePanel().

Exercise 3

Create 3 columns as space for your slider. **HINT**: Use column().

Exercise 4

Inside the column() you just created place a well Panel. **HINT**: Use wellPanel().

Exercise 5

In the well Panel you just created put a slider with features min = 10, max = 200, value = 40 and step = 10. **HINT**: Use sliderInput().

Now we are going to create the message that will explain why the plot will not be displayed when the JS expression is NOT TRUE.

Exercise 6

Use 5 columns as space for your message and the plot we are going to create later. The message will be “Less than 40-Error”.

Conditional Panel

The condition of the Conditional Panel is a JavaScript expression. Input values like input$n are accessed with dots, as in input.n

```
#ui.R
conditionalPanel("input.n >= 50")
```

Exercise 7

In the columns you just created place a conditional panel with condition the number of the input “c” to be more than 40 in order to display the plot.

Exercise 8

Create the ui side of your plot. **HINT**: Use plotOutput().

This is the server side of the scatter plot you want to create.

```
#server.R
output$sc <- renderPlot({
x <- rnorm(input$c)
y <- rnorm(input$c)
plot(x, y)
})
```

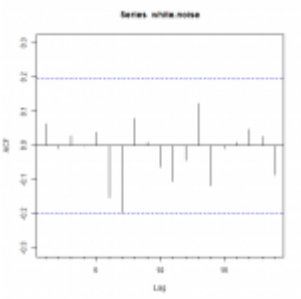
Exercise 9

Create the server side of your plot. **HINT**: Use renderPlot().

Exercise 10

Change your condition to <=40 and spot the differnece.

Shiny Applications Layouts Exercises (Part-6)



Shiny Applications Layouts – Absolutely-positioned panel

In the sixth part of our journey through Shiny App Layouts we will meet the absolutely-positioned panels. These are panels that you can drag and drop or not wherever you want in the interface. Moreover you can put anything in them, including inputs and outputs.

This part can be useful for you in two ways.

First of all, you can see different ways to enhance the appearance and the utility of your shiny app.

Secondly you can make a revision on what you learnt in "Building Shiny App" series as we will build basic shiny staff in order to present it in the proper way.

Read the examples below to understand the logic of what we are going to do and then test yous skills with the exercise set we prepared for you. Lets begin!

Answers to the exercises are available here.

Install Packages

For this app we will need the package markdown.

Exercise 1

Install and call the markdown package.

Build App

Exercise 2

Create a fluidpage with title “ABSOLUTE PANEL”. **HINT**: Use fluidpage().

In order to create this type of Panel you have to use the absolutePanel() function like the example below:

```
#ui.R
fluidPage(
h1("Absolutely"),
absolutePanel())
```

Exercise 3

Apply absolutePanel() function to your UI.

Exercise 4

Add a well Panel inside the absolutePanel(). **HINT**: Use wellPanel().

With the help of the markdown package we will add some random text like the example below:

```
#ui.R
HTML(markdownToHTML(fragment.only=TRUE, text=c(
"bla bla bla bla bla bla
bal blabla balalallalal
bla bla bla bla bla bla"
)))
```

Interactive Web Apps – Next Level Data Visualization. In this course you will learn how to create advanced Shiny web apps; embed video, pdfs and images; add focus and zooming tools; and many other functionalities (30 lectures, 3hrs.).

Exercise 5

Write a random text message using markdown and place it in your well Panel.

We will build an absolute Panel that uses bottom and right attributes. We also set draggable = TRUE, in order to move it.

```
#ui.R
absolutePanel(
bottom = 50, right = 50, width = 200,
draggable = TRUE)
```

Exercise 6

Practice with different values for bottom,left,right and width attributes and also set draggable to TRUE.

As already mentioned you can put anything you want in an abloslute Panel.For example:

```
#ui.R
absolutePanel(
sliderInput("s", "", min=3, max=20, value=5),
plotOutput("plot", height="100px"))
#server.R
function(input, output, session) {
output$plot <- renderPlot({
plot(head(cars, input$s), main="Cars")
})
}
```

Exercise 7

Put a slider in your absolute Panel. **HINT**: Use sliderInput().

Exercise 8

Then put a scatter plot like the one in the example above and connect it with your slider. **HINT**: Use plotOutput().

Fixed Panel

You can place your absolutePanel() function at the top of the screen using top, left, and right attributes.

Furthermore with fixed=TRUE, you can stabilize it.

```
#ui.R
absolutePanel(
top = 0, left = 0, right = 0,
fixed = TRUE)
```

Exercise 9

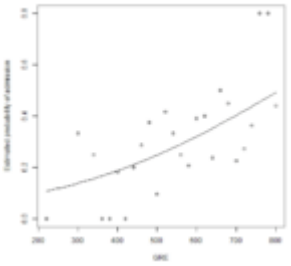
Place your absolute Panel to the top. Play with the parameters to understand how they affect its position.

Exercise 10

Stabilize your absolute Panel with fixed.

Shiny Application Layouts Exercises (Part-2)

SHINY APPLICATION LAYOUTS-PLOT PLUS COLUMNS



In the second part of our series we will build another small

shiny app but use another UI.

More specifically we will present the example of a UI with a plot at the top and columns at the bottom that contain the inputs that drive the plot. For our case we are going to use the diamonds dataset to create a Diamonds Analyzer App.

This part can be useful for you in two ways.

First of all, you can see different ways to enhance the appearance and the utility of your shiny app.

Secondly you can make a revision on what you learnt in the "Building Shiny App" series as we will build basic shiny stuff in order to present it in the proper way.

Read the examples below to understand the logic of what we are going to do and then test your skills with the exercise set we prepared for you. Lets begin!

Answers to the are available here.

Shiny Installation

In order to create the app we have to install and load the package shiny.

Exercise 1

Install and load shiny.

Grid Layout

The sidebarLayout uses Shiny's grid layout functions. Rows are created by the fluidRow function and include columns defined by the column function. Column widths should add up to 12 within a fluidRow.
The first parameter to the column function is it's width. You can also change the position of columns to decide the location of UI elements. You can put columns to the right by adding the offset parameter to the column function. Each unit of offset

increases the left-margin of a column by a whole column.

Now let’s begin to build our UI. First of all we will place the fluidpage with a title as below:

```
#ui.R
library(shiny)
shinyUI(fluidPage(
title = "Diamonds",
h4("Diamonds Analyzer")

))
#server.R
library(shiny)
function(input, output) {}
```

Exercise 2

Create the initial UI of your app and name it “Diamonds Analyzer”.

You can use the fluidrow function with the column function of width =2 inside of it like this:

```
#ui.R
library(shiny)
shinyUI(fluidPage(
title = "Diamonds",
h4("Diamonds Analyzer"),
fluidRow(column(2),
column(2),
)

))
```

Exercise 3

Create a fluidrow with two columns of width = 4 inside it. **NOTE**: Do not expect to see something yet.

Now it is time to fill these columns with some tools that will

help us determine the variables that we are going to use for our plot.
In the first 4 columns we will put a selectInput as the code below.

```
#ui.R
fluidRow(column(4,
h4("Variable X"),
selectInput('x', 'X', names(diamonds)))
```

Exercise 4

Put a selectInput in the first 4 columns of your UI. Name it “Variable X”. HINT:Use names to get the names of the dataset diamonds as inputs.

Now let’s move to the next four columns. We are going to put in there another selectInput and select the second of the dataset’s names as default. We are also going to see what offset does by setting it to 1 and then deactivating it again like the example below. You can use the code as it is or change the parameters given to understand the logic behind its use.

```
#ui.R
offset = 1,
selectInput('y', 'Y', names(dataset), names(dataset)[[2]])
```

Exercise 5

Create a selectInput from column 5 to column 8. Choose the second of the dataset’s name as default. Name it “Variable Y”. HINT: Use names to get the names of the dataset diamonds as inputs.

Exercise 6

Set the offset parameter to 1 from columns 5 to 8.

Now let’s call our plot and put it on the top of our UI. Look at the example below.

Exercise 7

Place the plot on the top of your UI. **HINT**: Use plotOutput and hr. **NOTE**: You are not going to see the plot in your UI because you have not created the server side yet.

We are going to create a reactive expression in order to combine the selected variables into a new data frame.Look at the example:

```
#server.R
selectedData <- reactive({
diamonds[, c(input$x, input$y)]
})
```

Exercise 8

Create a reactive expression in order to combine the selected variables into a new data frame. **HINT**: Use reactive.

Now plot your new data frame like the example:

```
#server.R
output$plot <- renderPlot({
plot(selectedData())
})
```

Exercise 9

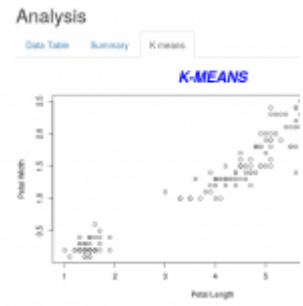
Plot your data frame. **HINT**: Use renderPlot.

As mentioned before the width of our UI is equal to 12 columns. So what is goint to happen if we a add a new column of width = 4 next to the other two? You have to find out in order to understand better how it works.

Exercise 10

Create a new selectInput and try to put it next to “Variable Y”. Try to explain the result. **NOTE**: You do not have to connect it with your plot.

Shiny Application Layouts Exercises (Part-1)



Shiny Application Layouts part 1

Welcome to the first part of our new series "Shiny Application Layouts". As you can understand from the title we will see how to organize the output of our application in various ways. For this reason we will build together 10 simple apps that will help you understand what kind of interfaces shiny provides.

In this part we will see tabsets which is one of the simplest ways to organize our app. This part can be useful for you in two ways.
First of all, as already mentioned, you can see different ways to enhance the appearance and the utility of your shiny app.
Secondly you can make a revision on what you learnt in Shiny Apps series as we will build basic shiny staff in order to present it in the proper way.

In part 1 we will use the dataset data which is loaded in R by default and we will create three tabPanel. Each one of them with its own utility.

Read the examples below to understand the logic of what we are going to do and then test yous skills with the exercise set we prepared for you. Lets begin!

Answers to the exercises are available here.

Tabsets

We use tabset in our application application to organize output. Firstly install the shiny package and then call it in a new r script of your working directory.
You can use install.packages() and library() for these.

Exercise 1

Install and load the shiny package.

Tab Panels

Tabsets can be created by calling the tabsetPanel function. Each tab panel includes a list of output elements.

In this case we will create a histogram, a summary and table view of the data, each rendered on their own tab.

Let's start by creating the basic interface like the example below: As we know the UI includes three basic parts (headerPanel, sidebarPanel, mainPanel).

```
#ui.R
library(shiny)

shinyUI(pageWithSidebar(
headerPanel("Example"),
sidebarPanel(),
mainPanel()
))
#server.R
shinyServer(function(input, output) {})
```

Exercise 2

Create the basic interface, name it "APP 1".

Secondly let's fill our sidebar with some widgets. We will create radioButtons and sliderInput to control the random

distribution and the number of observations. Look at the example.

```
#ui.R
library(shiny)

shinyUI(pageWithSidebar(
headerPanel("Example"),
sidebarPanel(
radioButtons("cho", "Choice:",
list("One" = "one",
"Two" = "two",
"Three" = "three",
"Four" = "four"
br(),

sliderInput("n",
"Number of observations:",
value = 50,
min = 1,
max = 100)
),
mainPanel()
))
#server.R
shinyServer(function(input, output) {})
```

Exercise 3

Put radioButtons inside sidebarPanel.The four choices will be “Normal”, “Uniform”, “Log-Normal”, “Exponential” and the name of it “Distributions”.

Exercise 4

Put sliderInput under radioButtons. Its values should be from 1 to 1000 and the selected value should be 500. Name it “Number of observations”.

It is time to create three tabsets in order to place there a

plot, summary, and table view of our dataset. More specifically we will use a plotOutput, a verbatimTextOutput and a tableOutput. Look at the code and the build your own.

```
#ui.R
tabsetPanel(
tabPanel("Plot", plotOutput("plot")),
tabPanel("Summary", verbatimTextOutput("summary")),
tabPanel("Table", tableOutput("table"))
)
```

Exercise 5

Create your tabsetPanel which will include the tabPanel.

Exercise 6

Create the first tabPanel for your Plot.Use plotOutput.
Create the second tabPanel for your Summary Statistics.Use verbatimTextOutput.
Create the third tabPanel for your Table.Use tableOutput.

Tabs and Reactive Data

Working with tabs into our user-interface magnifies the importance of creating reactive expressions for our data. In this example each tab provides its own view of the dataset. We have to mention that the bigger the dataset the slower our app will be.
We should use a Reactive expression to generate the requested distribution. This is called whenever the inputs change. The renderers are defined below then all use the value computed from this expression.

```
#server.R
```

```
data <- reactive({
dist <- switch(input$dist,
norm = rnorm,
unif = runif,
lnorm = rlnorm,
exp = rexp,
rnorm)

dist(input$n)
})
```

Exercise 7

Put data() in the correct place of your code in order to activate it.

Now we should generate a plot of the data. Note that the dependencies on the data are both tracked by the dependency graph.

As you can see from the code below we use the renderPlot function to create the reactive plot and inside there we use the hist function in order to create the histogram.

```
#server.R
output$plot <- renderPlot({
dist <- input$dist
n <- input$n

hist(data())
})
```

Exercise 8

Create a reactive histogram inside the tabPanel “Plot”

Let’s print a summary of the data in the tabPanel “Summary”. As you can see below we will use renderPrint for our case and the function summary.

```
#server.R
output$summary <- renderPrint({
```

```
summary(data())
})
```

Exercise 9

Print tha dataset's summary in the tabPanel "Summary".

Last but not least we will create an HTML table view of the data. For this we will use renderTable and data.frame function.

```
#server.R
output$table <- renderTable({
data.frame(x=data())
})
```

Exercise 10

Create a data table in the tabPanel "Table".

Building Shiny App exercises part 10

SHINY DASHBOARD STRUCTURE & APPEARANCE

Finally we reached in the final part of our series. At this part we will see how to improve the structure and the appearance of our dashboard even more, according to our

preferences and of course make it more attractive to the user. Last but not least we will see the simplest and easiest way to deploy it.

Read the examples below to understand the logic of what we are going to do and then test yous skills with the exercise set we prepared for you. Lets begin!

Answers to the exercises are available here.

infoBox

There is a special kind of box that is used for displaying simple numeric or text values, with an icon.The example code below shows how to generate infoBox. The first row of infoBox uses the default setting of fill=FALSE.

Since the content of an infoBox will usually be dynamic, shinydashboard contains the helper functions infoBoxOutput and renderInfoBox for dynamic content.

```
#ui.R
library(shinydashboard)

dashboardPage(
dashboardHeader(title = "Info boxes"),
dashboardSidebar(),
dashboardBody(
# infoBoxes with fill=FALSE
fluidRow(
# A static infoBox
infoBox("New Orders", 10 * 2, icon = icon("credit-card")),
# Dynamic infoBoxes
infoBoxOutput("progressBox"),
infoBoxOutput("approvalBox")
)))
#server.R
shinyServer(function(input, output) {
output$progressBox <- renderInfoBox({
infoBox(
```

```
"Progress", paste0(25 + input$count, "%"), icon =
icon("list"),
color = "purple"
)
})
output$approvalBox <- renderInfoBox({
infoBox(
"Approval", "80%", icon = icon("thumbs-up", lib =
"glyphicon"),
color = "yellow"
)
})
})
```

Exercise 1

Create three infoBox with information icons and color of your choice and put them in the tabItem “dt” under the “DATA-TABLE”.

To fill them with a color follow the example below:

```
#ui.R
library(shinydashboard)

dashboardPage(
dashboardHeader(title = "Info boxes"),
dashboardSidebar(),
dashboardBody(
# infoBoxes with fill=FALSE
fluidRow(
# A static infoBox
infoBox("New Orders", 10 * 2, icon = icon("credit-card"),fill
= TRUE),
# Dynamic infoBoxes
infoBoxOutput("progressBox"),
infoBoxOutput("approvalBox")
)))
#server.R
```

```
shinyServer(function(input, output) {
output$progressBox <- renderInfoBox({
infoBox(
"Progress", paste0(25 + input$count, "%"), icon =
icon("list"),
color = "purple",fill = TRUE
)
})
output$approvalBox <- renderInfoBox({
infoBox(
"Approval", "80%", icon = icon("thumbs-up", lib =
"glyphicon"),
color = "yellow",fill = TRUE
)
})
})
```

Exercise 2

Fill the infoBox with the color you selected in Exercise 1. HINT: Use fill.

Exercise 3

Now enhance the appearance of your tabItem named “km” by setting height = 450 in the four box you have there.

Skins

There are a number of color themes, or skins. The default is blue, but there are also black, purple, green, red, and yellow. You can choose which theme to use with dashboardPage(skin = "blue"), dashboardPage(skin = "black"), and so on.

Exercise 4

Change skin from blue to red.

CSS

You can add custom CSS to your app by adding code in the UI of your app like this:

```
#ui.R
dashboardPage(
dashboardHeader(title = "Custom font"),
dashboardSidebar(),
dashboardBody(
tags$head(tags$style(HTML('
.main-header .logo {

font-weight: bold;
font-size: 24px;
}
')))
)
)
```

Exercise 5

Change the font of your dashboard title by adding CSS code.

Long titles

In some cases, the title that you wish to use won’t fit in the default width in the header bar. You can make the space for the title wider with the titleWidth option. In this example, we’ve increased the width for the title to 450 pixels.

```
#ui.R
dashboardPage(
dashboardHeader(
title = "Example of a long title that needs more space",
titleWidth = 450
),
dashboardSidebar(),
dashboardBody(

)
)
```

```
#server.R
function(input, output) { }
```

Exercise 6

Set your titlewidth to “400” and then set it to the default value again.

Sidebar width

To change the width of the sidebar, you can use the width option. This example has a wider title and sidebar:

```
#ui.R
library(shinydashboard)
dashboardPage(
dashboardHeader(
title = "Title and sidebar 350 pixels wide",
titleWidth = 350
),
dashboardSidebar(
width = 350,
sidebarMenu(
menuItem("Menu Item")
)
),
dashboardBody()
)
#server.R
function(input, output) { }
```

Exercise 7

Set sidebar width to “400” and then return to the default one.

Icons

Icons are used liberally in shinydashboard. The icons used in shiny and shinydashboard are really just characters from special font sets, and they’re created with Shiny’s icon()

function.

To create a calendar icon, you'd call:
icon("calendar")

The icons are from Font-Awesome and Glyphicons. You can see lists of all available icons here:

http://fontawesome.io/icons/
http://getbootstrap.com/components/#glyphicons

Exercise 8

Change the icon of the three menuItem of your dashboard. Select whatever you like from the two lists above.

Statuses and colors

Many shinydashboard components have a status or color argument.

The status is a property of some Bootstrap classes. It can have values like status="primary", status="success", and others.

The color argument is more straightforward. It can have values like color="red", color="black", and others.

The valid statuses and colors are also listed in ?validStatuses and ?validColors.

Exercise 9

Change the status of the three widget box in the tabItem named "km" to "info", "success" and "danger" respectively.

Shinyapps.io

The easiest way to turn your Shiny app into a web page is to use shinyapps.io, RStudio's hosting service for Shiny apps.

shinyapps.io lets you upload your app straight from your R

session to a server hosted by RStudio. You have complete control over your app including server administration tools.

First of all you have to create an account in shinyapps.io.

Exercise 10

Publish your app through Shinyapps.io

Building Shiny App Exercises (part-9)

Shiny Dashboard Overview

In this part we will “dig deeper” to discover the amazing capabilities that a Shiny Dasboard provides.
Read the examples below to understand the logic of what we are going to do and then test yous skills with the exercise set we prepared for you. Lets begin!

Answers to the exercises are available here.

The dashboardPage function expects three components: a header, sidebar, and body:

```
#ui.R
dashboardPage(
dashboardHeader(),
```

```
dashboardSidebar(),
dashboardBody()
)
```

For more complicated apps, splitting app into pieces can make it more readable:

```
header <- dashboardHeader()

sidebar <- dashboardSidebar()

body <- dashboardBody()

dashboardPage(header, sidebar, body)
```

Now we'll look at each of the three main components of a shinydashboard.

HEADER

A header can have a title and dropdown menus. The dropdown menus are generated by the dropdownMenu function. There are three types of menus – messages, notifications, and tasks – and each one must be populated with a corresponding type of item.

Message menus

A messageItem contained in a message menu needs values for from and message. You can also control the icon and a notification time string. By default, the icon is a silhouette of a person. The time string can be any text. For example, it could be a relative date/time like "5 minutes", "today", or "12:30pm yesterday", or an absolute time, like "2014-12-01 13:45".

```
#ui.R
dropdownMenu(type = "messages",
messageItem(
from = "Sales Dept",
message = "Sales are steady this month."
```

```
),
messageItem(
from = "New User",
message = "How do I register?",
icon = icon("question"),
time = "13:45"
),
messageItem(
from = "Support",
message = "The new server is ready.",
icon = icon("life-ring"),
time = "2014-12-01"
)
)
```

Exercise 1

Create a dropdownMenu in your dashboardHeader as the example above. Put date, time and generally text of your choice.

Dynamic content

In most cases, you'll want to make the content dynamic. That means that the HTML content is generated on the server side and sent to the client for rendering. In the UI code, you'd use dropdownMenuOutput like this:

```
dashboardHeader(dropdownMenuOutput("messageMenu"))
```

Exercise 2

Replace dropdownMenu with dropdownMenuOutput and the three messageItem with messageMenu.

The next step is to create some messages for this example.The code below does this work for us.

```
# Example message data in a data frame
messageData <- data.frame(
from = c("Admininstrator", "New User", "Support"),
```

```
message = c(
"Sales are steady this month.",
"How do I register?",
"The new server is ready."
),
stringsAsFactors = FALSE
)
```

Exercise 3

Put messageData inside your server.r but outside of the shinyServer function.

And on the server side, you'd generate the entire menu in a renderMenu, like this:

```
output$messageMenu <- renderMenu({
# Code to generate each of the messageItems here, in a list.
messageData
# is a data frame with two columns, 'from' and 'message'.
# Also add on slider value to the message content, so that
messages update.
msgs <- apply(messageData, 1, function(row) {
messageItem(
from = row[["from"]],
message = paste(row[["message"]], input$slider)
)
})

dropdownMenu(type = "messages", .list = msgs)
})
```

Exercise 4

Put the code above(output$messageMenu) in the shinyServer of server.R.

Hopefully you have understood by now the logic behind the dynamic content of your Menu. Now let's return to the static one in order to describe it a little bit more. So make the

proper changes to your code in order to return exactly to the point we were after exercise 1.

Notification menus

A notificationItem contained in a notification contains a text notification. You can also control the icon and the status color. The code below gives an example.

```
#ui.r
dropdownMenu(type = "notifications",
notificationItem(
text = "20 new users today",
icon("users")
),
notificationItem(
text = "14 items delivered",
icon("truck"),
status = "success"
),
notificationItem(
text = "Server load at 84%",
icon = icon("exclamation-triangle"),
status = "warning"
)
)
```

Exercise 5

Create a dropdownMenu for your notifications like the example. Use text of your choice. Be careful of the type and the notificationItem.

Task menus

Task items have a progress bar and a text label. You can also specify the color of the bar. Valid colors are listed in ?validColors. Take a look at the example below.

```
#ui.r
dropdownMenu(type = "tasks", badgeStatus = "success",
```

```
taskItem(value = 90, color = "green",
"Documentation"
),
taskItem(value = 17, color = "aqua",
"Project X"
),
taskItem(value = 75, color = "yellow",
"Server deployment"
),
taskItem(value = 80, color = "red",
"Overall project"
)
)
```

Exercise 6

Create a dropdownMenu for your tasks like the example above. Use text of your choice and create as many taskItem as you want. Be carefull of the type and the taskItem.

Disabling the header

If you don’t want to show a header bar, you can disable it with:

```
dashboardHeader(disable = TRUE)
```

Exercise 7

Disable the header.

Now enable it again.

Body

The body of a dashboard page can contain any regular Shiny content. However, if you’re creating a dashboard you’ll likely want to make something that’s more structured. The basic building block of most dashboards is a box. Boxes in turn can contain any content.

Boxes

Boxes are the main building blocks of dashboard pages. A basic box can be created with the box function, and the contents of the box can be (most) any Shiny UI content. We have already created some boxes in part 8 so lets enhance theis appearance a little bit.
Boxes can have titles and header bar colors with the title and status options. Look at the examples below.

```
box(title = "Histogram", status = "primary",solidHeader = TRUE, plotOutput("plot2", height = 250)),
```

```
box(
title = "Inputs", status = "warning",
"Box content here", br(), "More box content",
sliderInput("slider", "Slider input:", 1, 100, 50),
textInput("text", "Text input:")
)
```

Exercise 8

Give a title of your choice to all the box you have created in your dashboard except of the three widgets' box.

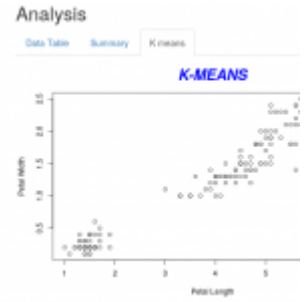
Exercise 9

Change the status of the first three box to "primary" and the last three to "warning".

Exercise 10

Transform the headers of your first three box to solid headers.

Building Shiny App Exercises (part-8)



Transform your App into Dashboard

Now that we covered the basic staff that you need to know in order to build your App it is time to enhance its appearance and its functionality. The interface is very important fot the user as it must not only be friendly but also easy to use.

At this part we will transform your Shiny App into a beautiful Shiny Dashboard. Firstly we will create the interface and then step by step we will “move” the App you built in the previous parts into this. In part 8 we will move the app step by step into your dashboard and in the last two parts we will enhance its appearance even more and of course deploy it.

Read the examples below to understand the logic of what we are going to do and then test yous skills with the exercise set we prepared for you. Lets begin!

Answers to the exercises are available here.

INSTALLATION

The packages that we are going to use is shinydashboard and shiny . To install, run:

```
install.packages("shinydashboard")
install.packages("shiny")
```

Learn more about Shiny in the online course R Shiny Interactive Web Apps – Next Level Data Visualization. In this course you will learn how to create advanced Shiny web apps; embed video, pdfs and images; add focus and zooming tools; and many other functionalities (30 lectures, 3hrs.).

Exercise 1

Install the package shinydashboard and the package shiny in your working directory.

BASICS

A dashboard has three parts: a header, a sidebar, and a body. Here's the most minimal possible UI for a dashboard page.

```
## ui.R ##
library(shinydashboard)

dashboardPage(
dashboardHeader(),
dashboardSidebar(),
dashboardBody()
)
```

Exercise 2

Add a dashboardPage and then Header, Sidebar and Body into your UI. HINT: Use dashboardPage, dashboardHeader, dashboardSidebar, dashboardBody.

First of all we should name it with title like below:

```
## ui.R ##
library(shinydashboard)

dashboardPage(
dashboardHeader(title="Dashboard),
dashboardSidebar(),
dashboardBody()
)
```

Exercise 3

Name your dashboard “Shiny App”. HINT: Use title.

Next, we can add content to the sidebar. For this example we’ll add menu items that behave like tabs. These function similarly to Shiny’s tabPanels: when you click on one menu item, it shows a different set of content in the main body.

There are two parts that need to be done. First, you need to add menuItems to the sidebar, with appropriate tabNames.

```
## Sidebar content
dashboardSidebar(
sidebarMenu(
menuItem("Dashboard", tabName = "dashboard", icon = icon("dashboard")),
menuItem("Widgets", tabName = "widgets", icon = icon("th"))
)
)
```

Exercise 4

Create three menuItem, name them “DATA TABLE”, “SUMMARY” and “K-MEANS” respectively. Make sure to use distict tabName for each one of them. The icon is of your choise. HINT: Use menuItem, tabName and icon.

In the body, add tabItems with corresponding values for tabName:

```
## Body content
dashboardBody(
tabItems(
tabItem(tabName = "dashboard",
h2("Dashboard"),
fluidRow(
box()
)
),
tabItem(tabName = "widgets",
```

```
h2("WIDGETS")
),
)
)
```

Exercise 5

Add tabItems in dashboardBody. Be sure to give the same tabName to each one to get them linked with your menuItem. HINT: Use tabItems, tabItem, h2.

Obviously, this dashboard isn’t very useful. We’ll need to add components that actually do something. In the body we can add boxes that have content.

Firstly let’s create a box for our dataTable in the tabItem with tabName “dt”.

```
## Body content
dashboardBody(
tabItems(
tabItem(tabName = "dashboard",
h2("Dashboard"),
fluidRow(
box()
)
),
tabItem(tabName = "widgets",
h2("WIDGETS")
),
)
)
```

Exercise 6

Specify the fluidrow and create a box inside the “DATA TABLE” tabItem. HINT: Use fluidrow and box.

Exercise 7

Do the same for the other two tabItem. Create one fluidrow and one box in the “SUMMARY” and another fluidrow with four boxin the “K-MEANS”.

Now just copy and paste the code below, which you used in part 7 to move your dataTable inside the “DATA TABLE” tabItem.

```
#ui.R
dataTableOutput("Table"),width = 400
#server.R
output$Table <- renderDataTable(
iris,options = list(
lengthMenu = list(c(10, 20, 30,-1),c('10','20','30','ALL')),
pageLength = 10))
```

Exercise 8

Place the sample code above in the right place in order to add the dataTable “Table” inside the “DATA TABLE” tabItem.

Now just copy and paste the code below, which you used in part 7 to move the dataTable “Table2” inside the “SUMMARY” tabItem.

```
#ui.R
dataTableOutput("Table2"),width = 400

#server.R
sumiris<-as.data.frame.array(summary(iris))
output$Table2 <- renderDataTable(sumiris)
```

Exercise 9

Place the sample code above in the right place in order to add the dataTable “Table2” inside the “SUMMARY” tabItem.

Do the same for the last exercise as you just have to put the code from part 7 inside the “K-MEANS” tabItem.

Exercise 10

Place the K-Means plot and the three widgets from part 7 inside the four box you created before.