We all know that Shiny is great for interactive data visualisations. But, sometimes even the best attempts to fit all your graphs just in one Shiny page are not enough. From our experience, almost every project with growing amount of KPIs struggles with a problem of messy and not readable final reports. Here is where dashboards appear to be handy. Dashboards allow you to intuitively structure your reports by breaking them down into the sections, panels and tabs. Thanks to that it is much easier for a final user to navigate through your work.

shinydashboard does a decent job here. However, when you create a bunch of apps using it, you quickly realize that they all look the same and are simply boring. In this tutorial, I will show you how to take advantage of semantic.dashboard [package](https://appsilondatascience.com/blog/rstats/2018/04/23/semanticdashboard.html). This is an alternative to shinydashboard which makes use of [Semantic UI](https://semantic-ui.com/). Thanks to that you can introduce beautiful Semantic components into your app and select from many available themes.

**How to start?**

Let’s begin with creating an empty dashboard:

library(shiny)

library(semantic.dashboard)

ui <- dashboardPage(

dashboardHeader(),

dashboardSidebar(),

dashboardBody()

)

server <- shinyServer(function(input, output, session) {

})

shinyApp(ui, server)

For comparison you might check what happens if you change library(semantic.dashboard) to library(shinydashboard).

What you should see is something like this:

With almost no effort we have created the skeleton for our first semantic.dashboard app.

Now it is time to discuss basic components of a dashboard. Each dashboardPage consists of three elements:

* header,
* sidebar,
* body.

Currently our header is quite boring, so let’s add some title and change its colour:

dashboardHeader(color = "blue", title = "Dashboard Demo", inverted = TRUE)

We don’t need a sidebar to be so wide and to make it more functional let’s add two menu elements:

dashboardSidebar(

size = "thin", color = "teal",

sidebarMenu(

menuItem(tabName = "main", "Main"),

menuItem(tabName = "extra", "Extra")

)

)

This is the result you should expect to see:

Not bad! We can do even better by adding some icons to our menuItems. Can you do it yourself?

Hint! Use semantic.dashboard documentation, eg. by typing ?menuItem in RStudio console.

**Adding content**

In this section we will start filling our app with some content. We will use popular dataset [“mtcars”](https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/mtcars.html), extracted from the 1974 Motor Trend US magazine, which comprises some parameters of 32 old-school automobiles. Before that, let’s make sure that we know how to change the content of our body using tabs. Look at the following piece of code:

dashboardBody(

tabItems(

selected = 1,

tabItem(

tabName = "main",

fluidRow(

h1("main")

)

),

tabItem(

tabName = "extra",

fluidRow(

h1("extra")

)

)

)

)

We created two tabItems with tabNames exactly the same as menuItems. The selected parameters of tabItems tell us which tabItem should be selected and displayed after running the app.

Equipped with that knowledge, we can finally implement something really functional. We start from creating a simple plot describing the relationship between the car’s gearbox type and their miles per gallon parameter. In the shinyServer function we call:

data("mtcars")

mtcars$am <- factor(mtcars$am,levels=c(0,1),

labels=c("Automatic","Manual"))

output$boxplot1 <- renderPlot({

ggplot(mtcars, aes(x = am, y = mpg)) +

geom\_boxplot(fill = semantic\_palette[["green"]]) +

xlab("gearbox") + ylab("Miles per gallon")

})

Since we are using ggplot2 don’t forget to attach this package at the beginning of the script.

We are going to make another plot from this dataset, so let’s divide the main page into two sections. We can exchange content of fluidRow function of our first tabItem by:

box(width = 8,

title = "Graph 1",

color = "green", ribbon = TRUE, title\_side = "top right",

column(8,

plotOutput("boxplot1")

)

),

box(width = 8,

title = "Graph 2",

color = "red", ribbon = TRUE, title\_side = "top right"

)

This is a good moment to make sure that your app is running. If the answer is yes, proceed to the next section. In case it’s not, verify if you followed all the previous steps.

**Interactive items**

What do you say about adding more interactive plot to our dashboard? Let’s use [“plotly”](https://plot.ly/) to present the relation between the weight of a car and miles per gallon. I decided for a point plot here since I can include on it some extra information. By colour, I mark the number of cylinders and by the size of dots a quarter mile time. That’s the code to achieve it:

colscale <- c(semantic\_palette[["red"]], semantic\_palette[["green"]], semantic\_palette[["blue"]])

output$dotplot1 <- renderPlotly({

ggplotly(ggplot(mtcars, aes(wt, mpg))

+ geom\_point(aes(colour=factor(cyl), size = qsec))

+ scale\_colour\_manual(values = colscale)

)

})

Note that we used semantic\_palette here for graph colors in order to stay consistent with SemnaticUI layout of the app.

To insert it into the dashboard, we add by analogy to the second box:

box(width = 8,

title = "Graph 2",

color = "red", ribbon = TRUE, title\_side = "top right",

column(width = 8,

plotlyOutput("dotplot1")

)

)

That’s how our main page looks right now:

I would say that’s enough here. Let’s switch to another tab.

Probably, some curious user would like to see a complete list of cars properties in the “Extra” tab. Let’s give him that chance. For that we will use the DT package to render the table on the server side:

output$gstatstable <- DT::renderDataTable(github.data)

and display it in the second tab:

tabItem(

tabName = "extra",

fluidRow(

dataTableOutput("carstable")

)

)

I’m pretty satisfied of what we have achieved so far…

But you know what? I changed my mind at this point and decided to modify the theme of this dashboard. I reviewed my options on [Semantic Forest](http://semantic-ui-forest.com/themes/) website and decided for cerulean. It’s very easy to change the theme now, as it requires changing only one parameter in the dashboardPage function.

dashboardPage(

header, sidebar, body,

theme = "cerulean")

Let’s see how this app works:

**Full code**

Okay, so that’s it! If you got to this point, it means that you created your first dashboard with semantic.dashboard. For training, I encourage you to customize it by adding more tabs with new fancy plots. If you missed some step, here you can check the complete code for the dashboard:

library(shiny)

library(semantic.dashboard)

library(ggplot2)

library(plotly)

library(DT)

ui <- dashboardPage(

dashboardHeader(color = "blue",title = "Dashboard Demo", inverted = TRUE),

dashboardSidebar(

size = "thin", color = "teal",

sidebarMenu(

menuItem(tabName = "main", "Main", icon = icon("car")),

menuItem(tabName = "extra", "Extra", icon = icon("table"))

)

),

dashboardBody(

tabItems(

selected = 1,

tabItem(

tabName = "main",

fluidRow(

box(width = 8,

title = "Graph 1",

color = "green", ribbon = TRUE, title\_side = "top right",

column(width = 8,

plotOutput("boxplot1")

)

),

box(width = 8,

title = "Graph 2",

color = "red", ribbon = TRUE, title\_side = "top right",

column(width = 8,

plotlyOutput("dotplot1")

)

)

)

),

tabItem(

tabName = "extra",

fluidRow(

dataTableOutput("carstable")

)

)

)

), theme = "cerulean"

)

server <- shinyServer(function(input, output, session) {

data("mtcars")

colscale <- c(semantic\_palette[["red"]], semantic\_palette[["green"]], semantic\_palette[["blue"]])

mtcars$am <- factor(mtcars$am,levels=c(0,1),

labels=c("Automatic","Manual"))

output$boxplot1 <- renderPlot({

ggplot(mtcars, aes(x = am, y = mpg)) +

geom\_boxplot(fill = semantic\_palette[["green"]]) +

xlab("gearbox") + ylab("Miles per gallon")

})

output$dotplot1 <- renderPlotly({

ggplotly(ggplot(mtcars, aes(wt, mpg))

+ geom\_point(aes(colour=factor(cyl), size = qsec))

+ scale\_colour\_manual(values = colscale)

)

})

output$carstable <- renderDataTable(mtcars)

})

shinyApp(ui, server)