Applications of Data Science

Intro to Building Data Apps

Applications of Data Science - Class Bonus

Giora Simchoni

`gsimchoni@gmail.com and add #dsapps in subject`

Stat. and OR Department, TAU

2022-03-26

APPLICATIONS

OF DATA SCIENCE

Shiny in Four Apps

APPLICATIONS OF DATA SCIENCE

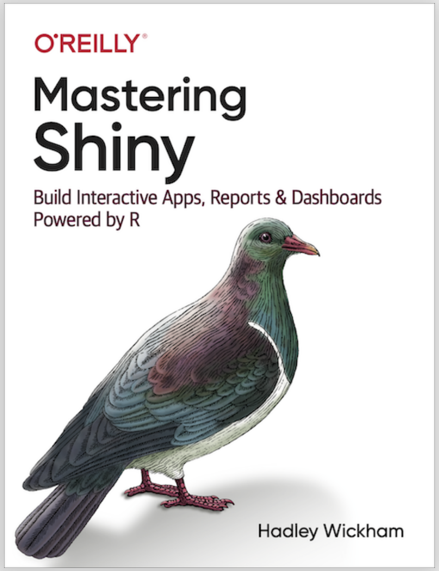
Shiny

Shiny is made in RStudio.

Start with the docs.

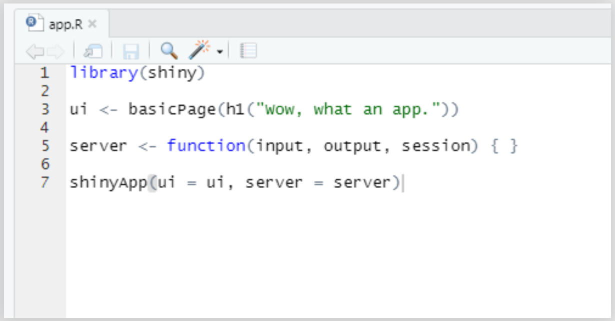
Or go to Zev Ross 40 (!) apps tutorial.

Or straight to God Himself:



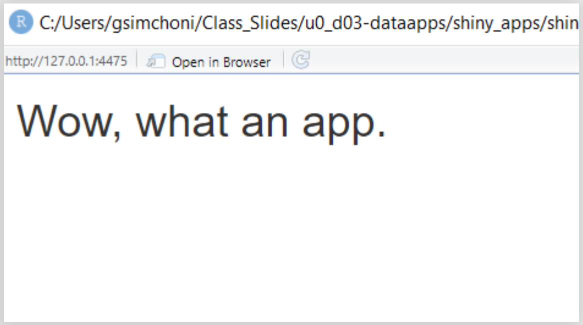
shiny01

A single `app.R` file containing your frontend (`ui`) and backend (`server`):



app.R

```r
library(shiny)

ui <- basicPage(h1("Wow, what an app."))

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

shinyApp(ui = ui, server = server)
```



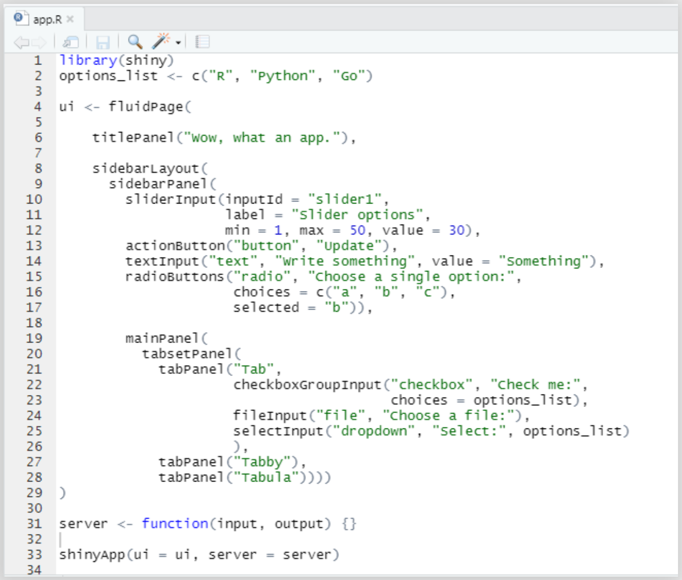
C:/Users/gsimchoni/Class_Slides/u0_d03-dataapps/shiny_apps/shin

http://127.0.0.1:4475

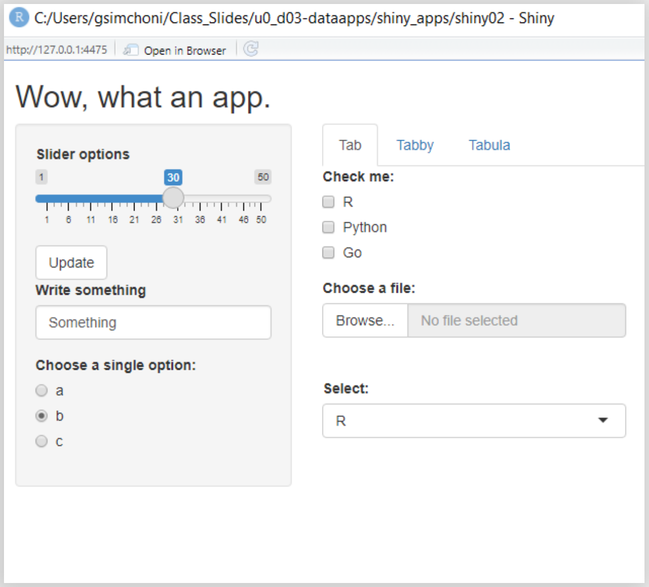
Wow, what an app.

shiny02

I recommend befriending the frontend (`ui`) first:



```r
library(shiny)
options_list <- c("R", "Python", "Go")

ui <- fluidPage(

    titlePanel("Wow, what an app."),

    sidebarLayout(
      sidebarPanel(
        sliderInput(inputId = "slider1",
                    label = "Slider options",
                    min = 1, max = 50, value = 30),
        actionButton("button", "Update"),
        textInput("text", "Write something", value = "Something"),
        radioButtons("radio", "Choose a single option:",
                     choices = c("a", "b", "c"),
                     selected = "b")),

        mainPanel(
          tabsetPanel(
            tabPanel("Tab",
                     checkboxGroupInput("checkbox", "Check me:",
                                        choices = options_list),
                     fileInput("file", "Choose a file:"),
                     selectInput("dropdown", "Select:", options_list)
                     ),
            tabPanel("Tabby"),
            tabPanel("Tabula"))))
)

server <- function(input, output) {}

shinyApp(ui = ui, server = server)
```

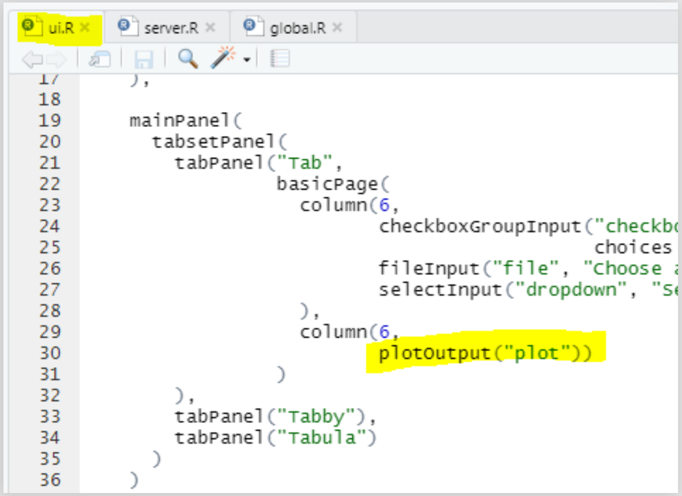


shiny03

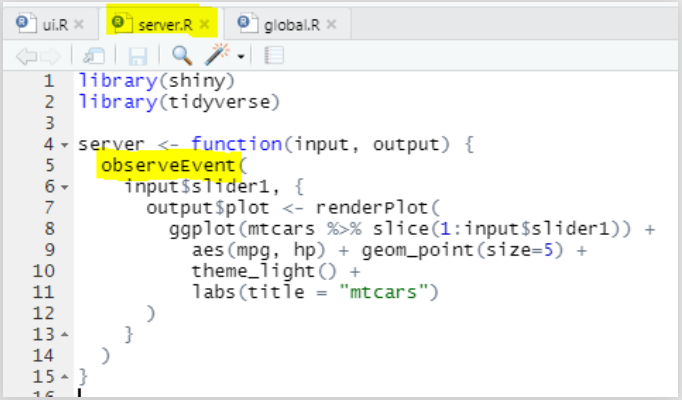
Once it becomes too much we go modular.

Backend (`server.R`) is where R does her thing.

`observeEvent()` of slider changing to re-render a plot:



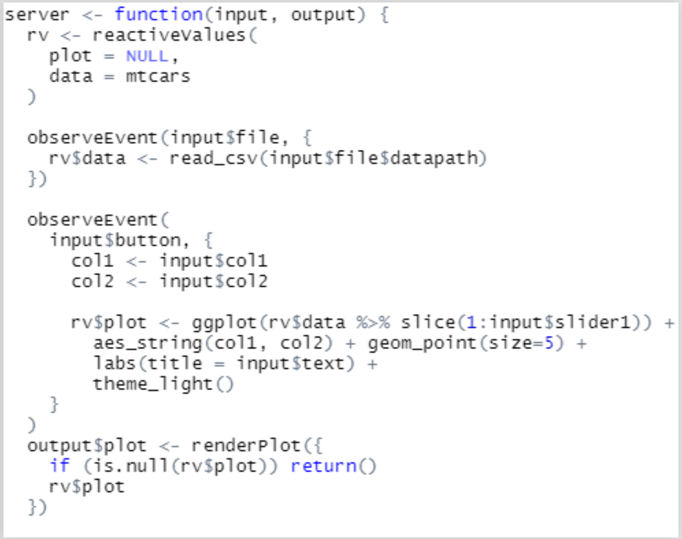
```
    ),

    mainPanel(
      tabsetPanel(
        tabPanel("Tab",
                 basicPage(
                   column(6,
                          checkboxGroupInput("checkbo
                                             choices
                          fileInput("file", "Choose a
                          selectInput("dropdown", "Se
                   ),
                   column(6,
                          plotOutput("plot"))
                 )
        ),
        tabPanel("Tabby"),
        tabPanel("Tabula")
      )
    )
```



```
library(shiny)
library(tidyverse)

server <- function(input, output) {
  observeEvent(
    input$slider1, {
      output$plot <- renderPlot(
        ggplot(mtcars %>% slice(1:input$slider1)) +
          aes(mpg, hp) + geom_point(size=5) +
          theme_light() +
          labs(title = "mtcars")
      )
    }
  )
}
```

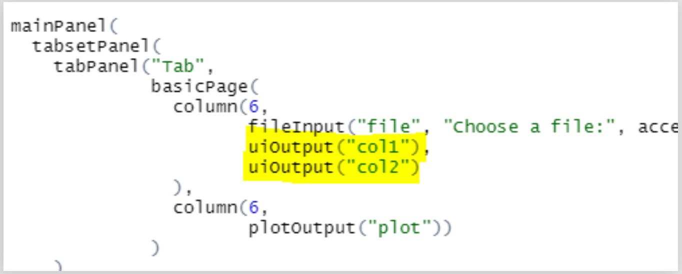
shiny04

Use `reactiveValues()` to keep the state of dynamic objects:

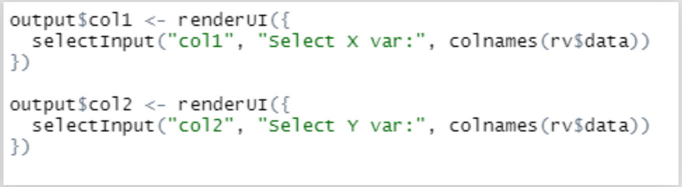


```
server <- function(input, output) {
  rv <- reactiveValues(
    plot = NULL,
    data = mtcars
  )

  observeEvent(input$file, {
    rv$data <- read_csv(input$file$datapath)
  })

  observeEvent(
    input$button, {
      col1 <- input$col1
      col2 <- input$col2

      rv$plot <- ggplot(rv$data %>% slice(1:input$slider1)) +
        aes_string(col1, col2) + geom_point(size=5) +
        labs(title = input$text) +
        theme_light()
    }
  )
  output$plot <- renderPlot({
    if (is.null(rv$plot)) return()
    rv$plot
  })
```

Use `renderUI()` for dynamic UI:



```
mainPanel(
  tabsetPanel(
    tabPanel("Tab",
             basicPage(
               column(6,
                      fileInput("file", "Choose a file:", acce
                      uiOutput("col1"),
                      uiOutput("col2")
               ),
               column(6,
                      plotOutput("plot"))
             )
    )
```



```
output$col1 <- renderUI({
  selectInput("col1", "Select X var:", colnames(rv$data))
})

output$col2 <- renderUI({
  selectInput("col2", "Select Y var:", colnames(rv$data))
})
```

Is that it?

Formulan

If you really want to be amazed

Visit the annual RStudio Shiny contest and the Shiny gallery.

Dash in Four Apps

Applications of Data Science

Dash

Dash is made by Plotly, other than Python it works with R and Julia.

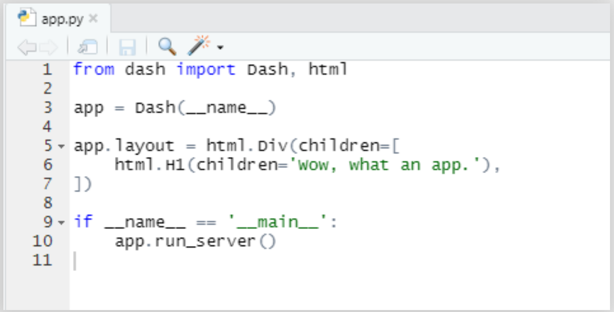
It is much "closer" to JavaScript (advantage?)

Start with the docs.

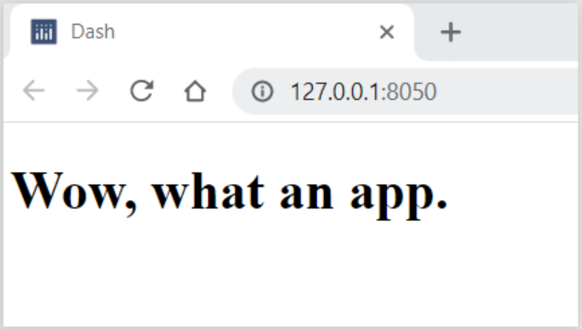
Another promising option is Voila by Jupyter.

dash01

A single `app.py` file containing your frontend (`layout`) and backend (`callbacks`):



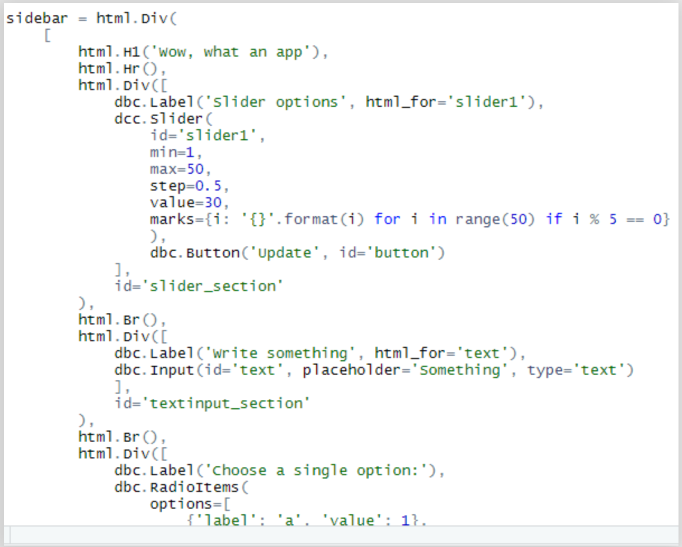
```python
from dash import Dash, html

app = Dash(__name__)

app.layout = html.Div(children=[
    html.H1(children='Wow, what an app.'),
])

if __name__ == '__main__':
    app.run_server()
```



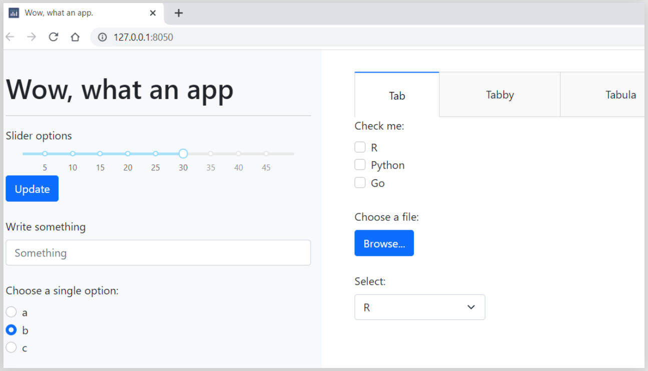
Wow, what an app.

dash02

I recommend befriending the frontend (`layout`) first:



```
sidebar = html.Div(
    [
        html.H1('Wow, what an app'),
        html.Hr(),
        html.Div([
            dbc.Label('Slider options', html_for='slider1'),
            dcc.Slider(
                id='slider1',
                min=1,
                max=50,
                step=0.5,
                value=30,
                marks={i: '{}'.format(i) for i in range(50) if i % 5 == 0}
                ),
                dbc.Button('Update', id='button')
            ],
            id='slider_section'
        ),
        html.Br(),
        html.Div([
            dbc.Label('Write something', html_for='text'),
            dbc.Input(id='text', placeholder='Something', type='text')
            ],
            id='textinput_section'
        ),
        html.Br(),
        html.Div([
            dbc.Label('Choose a single option:'),
            dbc.RadioItems(
                options=[
                    {'label': 'a', 'value': 1},
```

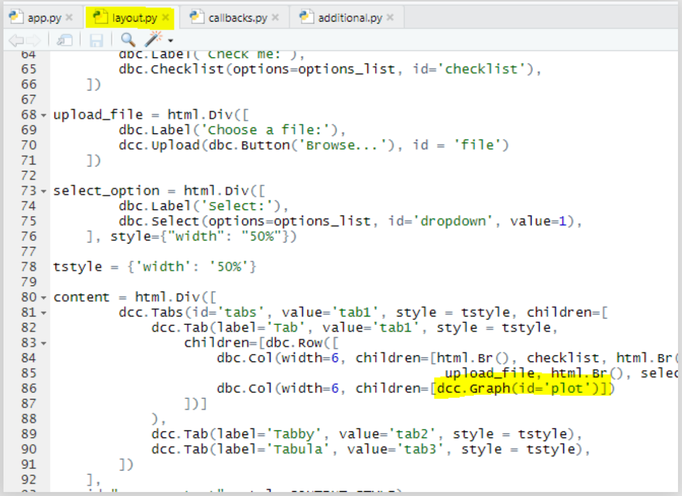


dash03

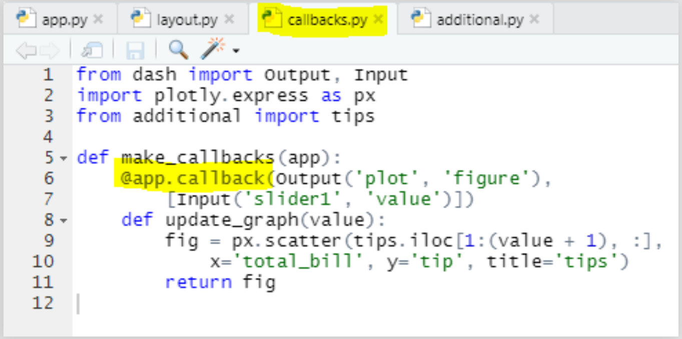
Once it becomes too much we go modular.

Backend (`callbacks.py`) is where Python does her thing.

`@app.callback()` of slider changing to re-render a plot:



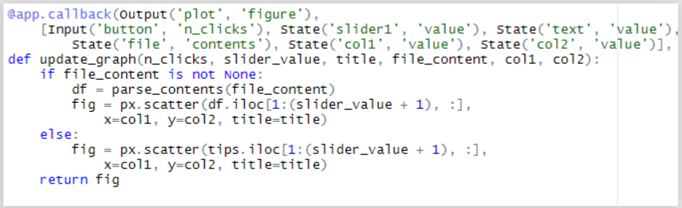
```python
        dbc.Label('Check me:'),
        dbc.Checklist(options=options_list, id='checklist'),
    ])

upload_file = html.Div([
        dbc.Label('Choose a file:'),
        dcc.Upload(dbc.Button('Browse...'), id = 'file')
    ])

select_option = html.Div([
        dbc.Label('Select:'),
        dbc.Select(options=options_list, id='dropdown', value=1),
    ], style={"width": "50%"})

tstyle = {'width': '50%'}

content = html.Div([
        dcc.Tabs(id='tabs', value='tab1', style = tstyle, children=[
            dcc.Tab(label='Tab', value='tab1', style = tstyle,
                children=[dbc.Row([
                    dbc.Col(width=6, children=[html.Br(), checklist, html.Br(
                                                    upload_file, html.Br(), selec
                    dbc.Col(width=6, children=[dcc.Graph(id='plot')])
                ])]
            ),
            dcc.Tab(label='Tabby', value='tab2', style = tstyle),
            dcc.Tab(label='Tabula', value='tab3', style = tstyle),
        ])
    ],
```



```python
from dash import Output, Input
import plotly.express as px
from additional import tips

def make_callbacks(app):
    @app.callback(Output('plot', 'figure'),
        [Input('slider1', 'value')])
    def update_graph(value):
        fig = px.scatter(tips.iloc[1:(value + 1), :],
            x='total_bill', y='tip', title='tips')
        return fig
```

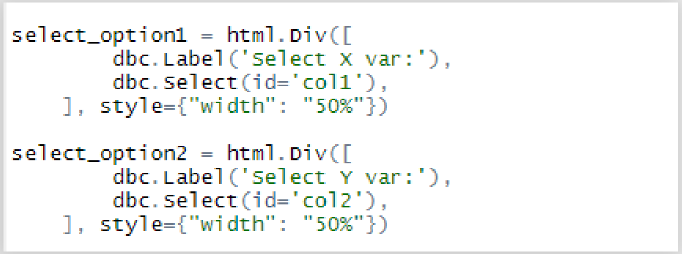
dash04

There are no `reactiveValues` in Dash backend (AFAIK), but we can do multiple `Output`s/`Input`s and `State`s

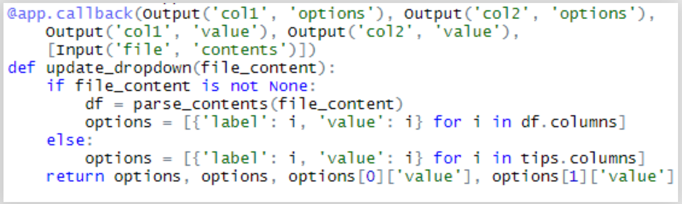


```
@app.callback(Output('plot', 'figure'),
    [Input('button', 'n_clicks'), State('slider1', 'value'), State('text', 'value'),
        State('file', 'contents'), State('col1', 'value'), State('col2', 'value')],
def update_graph(n_clicks, slider_value, title, file_content, col1, col2):
    if file_content is not None:
        df = parse_contents(file_content)
        fig = px.scatter(df.iloc[1:(slider_value + 1), :],
            x=col1, y=col2, title=title)
    else:
        fig = px.scatter(tips.iloc[1:(slider_value + 1), :],
            x=col1, y=col2, title=title)
    return fig
```

And rendering UI is very easy because every object's components are modifiable:



```
select_option1 = html.Div([
        dbc.Label('Select X var:'),
        dbc.Select(id='col1'),
    ], style={"width": "50%"})

select_option2 = html.Div([
        dbc.Label('Select Y var:'),
        dbc.Select(id='col2'),
    ], style={"width": "50%"})
```



```
@app.callback(Output('col1', 'options'), Output('col2', 'options'),
    Output('col1', 'value'), Output('col2', 'value'),
    [Input('file', 'contents')])
def update_dropdown(file_content):
    if file_content is not None:
        df = parse_contents(file_content)
        options = [{'label': i, 'value': i} for i in df.columns]
    else:
        options = [{'label': i, 'value': i} for i in tips.columns]
    return options, options, options[0]['value'], options[1]['value']
```

Is that it?

If you really want to be amazed

Visit the Dash gallery.

Dockerize your app!

Summary

Do I think you can replace the Front-end engineer at your organization? No.

But you can certainly use data apps for:

- Inside dashboards (everyone can access via company server or with Docker: Vivian)
- Personal tools (RateImagesApp, Formulan)
- Quick prototypes
- Showing people in company how data/analysis looks like and letting them playing with it
- Simulations
- Model testing