

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

2022-01-03

APPLICATIONS



OF DATA SCIENCE

Shiny in Four Apps

APPLICATIONS



OF DATA SCIENCE

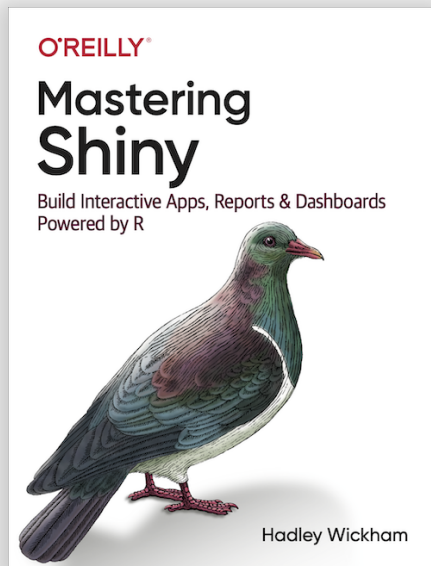
Shiny

Shiny is made in RStudio.

Start with the [docs](#).

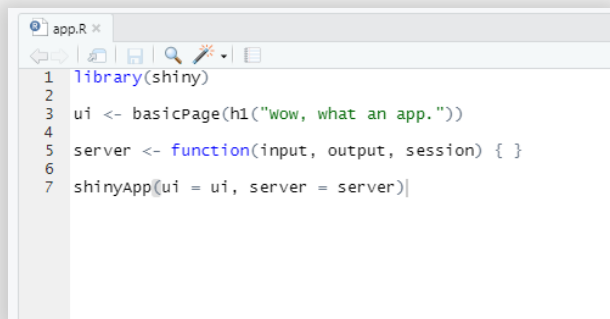
Or go to Zev Ross 40 (!) apps [tutorial](#).

Or straight to God Himself:

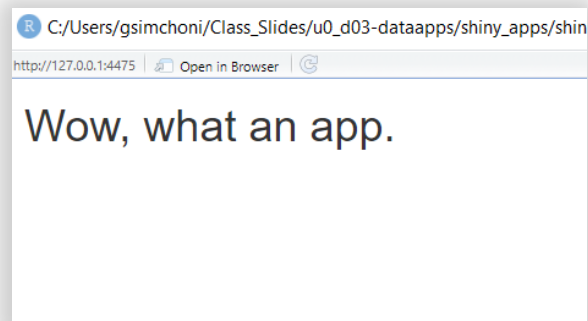


shiny01

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



```
1 library(shiny)
2
3 ui <- basicPage(h1("wow, what an app."))
4
5 server <- function(input, output, session) { }
6
7 shinyApp(ui = ui, server = server)
```



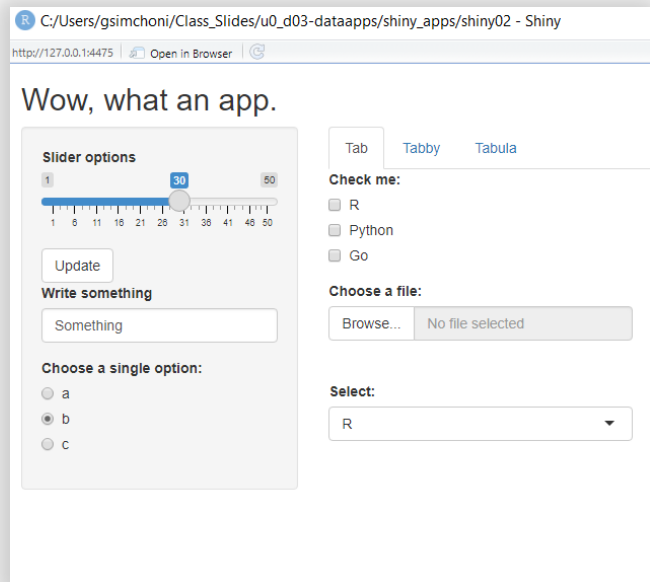
C:/Users/gsimchoni/Class_Slides/u0_d03-dataapps/shiny_apps/shin
http://127.0.0.1:4475 | Open in Browser

Wow, what an app.

shiny02

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

```
1 library(shiny)
2 options_list <- c("R", "Python", "Go")
3
4 ui <- fluidPage(
5
6   titlePanel("wow, what an app."),
7
8   sidebarLayout(
9     sidebarPanel(
10       sliderInput(inputId = "slider1",
11                  label = "Slider options",
12                  min = 1, max = 50, value = 30),
13       actionButton("button", "Update"),
14       textInput("text", "Write something", value = "Something"),
15       radioButtons("radio", "Choose a single option:",
16                   choices = c("a", "b", "c"),
17                   selected = "b"),
18
19       mainPanel(
20         tabsetPanel(
21           tabPanel("Tab",
22                   checkboxGroupInput("checkbox", "Check me:",
23                                     choices = options_list),
24                   fileInput("file", "choose a file:"),
25                   selectInput("dropdown", "Select:", options_list)
26                   ),
27           tabPanel("Tabby"),
28           tabPanel("Tabula")
29         )
30       )
31     )
32
33 server <- function(input, output) {}
34 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:

```
17
18
19   mainPanel(
20     tabsetPanel(
21       tabPanel("Tab",
22         basicPage(
23           column(6,
24             checkboxGroupInput("checkbox",
25               choices = choices,
26               fileInput("file", "Choose a file",
27                 selectInput("dropdown", "Select",
28                   ), column(6,
29                     plotOutput("plot"))
30                   ),
31                 ),
32             ),
33       tabPanel("Tabby"),
34       tabPanel("Tabula")
35     )
36   )
```

```
1 library(shiny)
2 library(tidyverse)
3
4 server <- function(input, output) {
5   observeEvent(
6     input$slider1, {
7     output$plot <- renderPlot(
8       ggplot(mtcars %>% slice(1:input$slider1)) +
9         aes(mpg, hp) + geom_point(size=5) +
10        theme_light() +
11        labs(title = "mtcars")
12     )
13   }
14 }
15 }
```

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:", accept = "text/csv"),  
          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?)

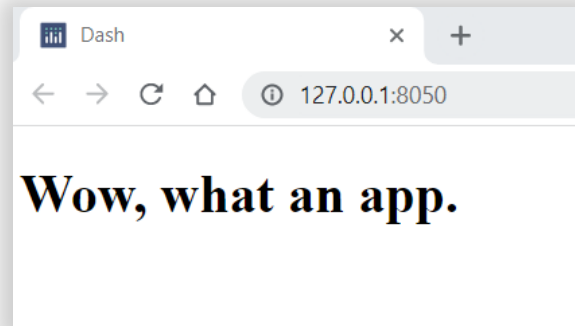
Start with the [docs](#).

Another promising option is [Voila](#) by Jupyter.

dash01

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

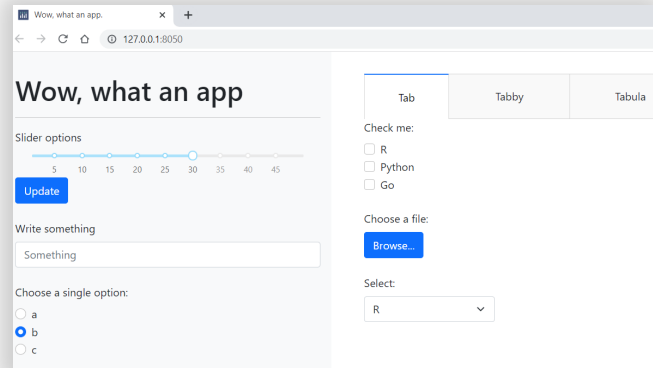
```
app.py x
1 from dash import Dash, html
2
3 app = Dash(__name__)
4
5 app.layout = html.Div(children=[
6     html.H1(children='wow, what an app.'),
7 ])
8
9 if __name__ == '__main__':
10     app.run_server()
11
```



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},
                {'label': 'b', 'value': 2},
                {'label': 'c', 'value': 3}
            ]
        )
    ])
])
```



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:

```
app.py x layout.py x callbacks.py x additional.py x
65 dbc.Label('Check me: '),
66 dbc.Checklist(options=options_list, id='checklist'),
67 ])
68 - upload_file = html.Div([
69     dbc.Label('Choose a file:'),
70     dcc.Upload(dbc.Button('Browse...'), id = 'file')
71 ])
72
73 - select_option = html.Div([
74     dbc.Label('Select:'),
75     dbc.Select(options=options_list, id='dropdown', value=1),
76 ], style={'width': '50%'})
77
78 tstyle = {'width': '50%'}
79
80 - content = html.Div([
81     dcc.Tabs(id='tabs', value='tab1', style = tstyle, children=[
82         dcc.Tab(label='Tab1', value='tab1', style = tstyle,
83             children=[dbc.Row([
84                 dbc.Col(width=6, children=[html.Br(), checklist, html.Br(),
85                     upload_file, html.Br(), select_option])
86                 dbc.Col(width=6, children=[dcc.Graph(id='plot')])
87             ])],
88         ),
89         dcc.Tab(label='Tabby', value='tab2', style = tstyle),
90         dcc.Tab(label='Tabula', value='tab3', style = tstyle),
91     ])
92 ],
93 style={'width': '100%'})
```

```
app.py x layout.py x callbacks.py x additional.py x
1 from dash import Output, Input
2 import plotly.express as px
3 from additional import tips
4
5 def make_callbacks(app):
6     @app.callback(Output('plot', 'figure'),
7                   [Input('slider1', 'value')])
8     def update_graph(value):
9         fig = px.scatter(tips.iloc[1:(value + 1), :],
10                         x='total_bill', y='tip', title='tips')
11         return fig
12
```


dash04

There are no
reactiveValues in Dash
backend (AFAIK), but we can do
multiple Outputs/Inputs and
States

```
@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](#).

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