

Getting to Know Shiny

Dr. Kam Tin Seong
Assoc. Professor of Information Systems

School of Computing and Information Systems,
Singapore Management University

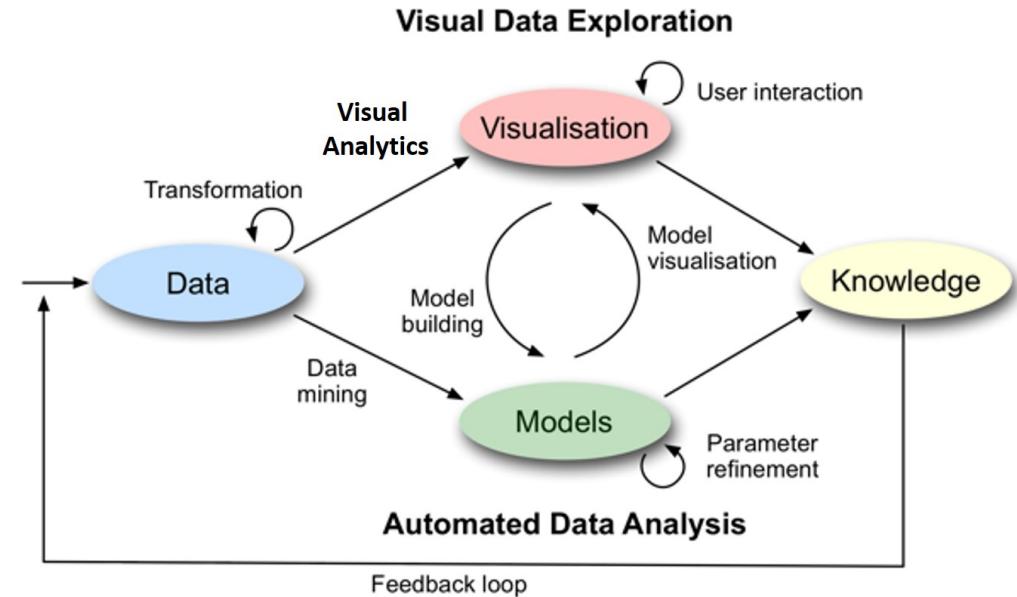
2019/11/25 (updated: 2021-07-17)

Content

- What is a Web-enabled Visual Analytics Application?
- Why building Web-enabled Visual Analytical Application?
- Evolution of web-based Technology
- Getting to Know Shiny

What is a Web-enabled Visual Analytics Application?

- Focuses and emphasises on **interactivity** and effective integration of techniques from **data analytics**, **visualization** and **human-computer interaction (HCI)**.



Why building a Web-enabled Visual Analytics Application?

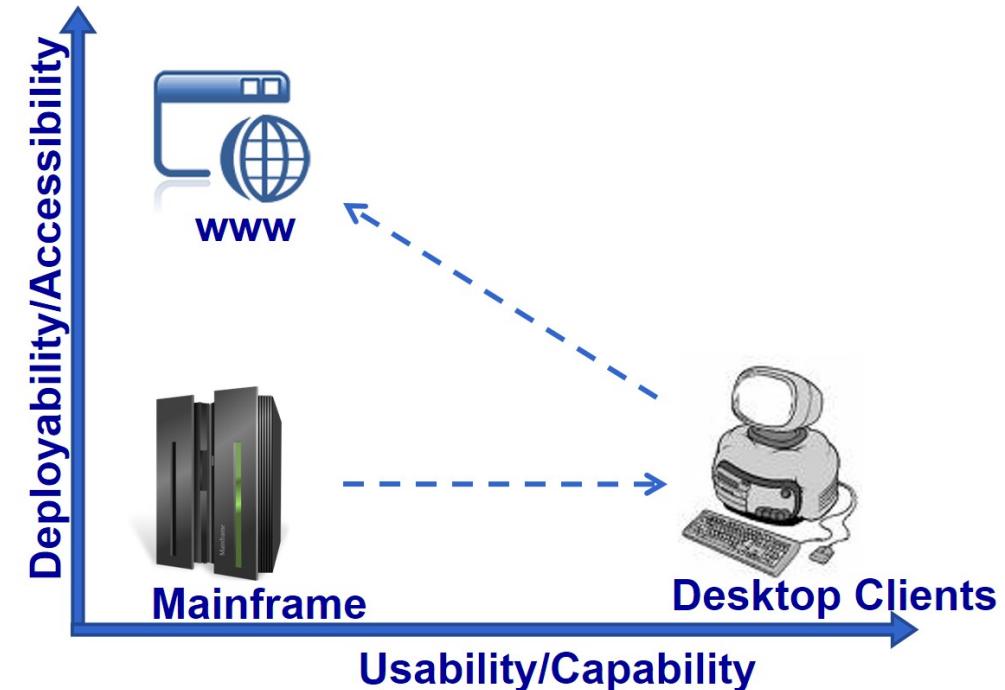
- To explore how the best of these different but related domains can be combined such that **the sum is greater than the parts**.
- To **democratise data and analytics** through web-based analytical applications for data exploration, visualisation analysis and modelling.



Source: <https://www.rekener.com/blog/democratize-data-analytics-customer-data-platform-B2B>

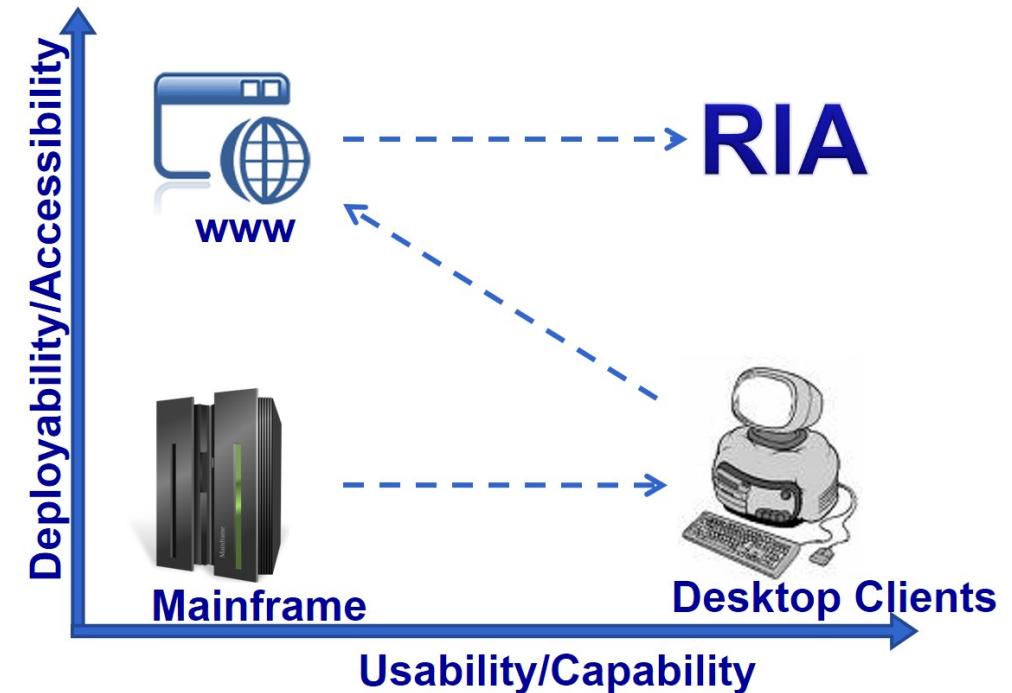
Technology Challenges

- Mainframe computing tend to have low usability and low accessibility.
- Desktop computing tend to have high usability but low accessibility.
- Web-based computing (including mobile computing) are highly accessible but with relatively low capability



Web-based data visualisation

- The break-through is Rich Internet Applications (RIA)



Reference: <https://www.computerworld.com/article/2551058/rich-internet-applications.html>

Development of RIA

First generation RIA data visualisation

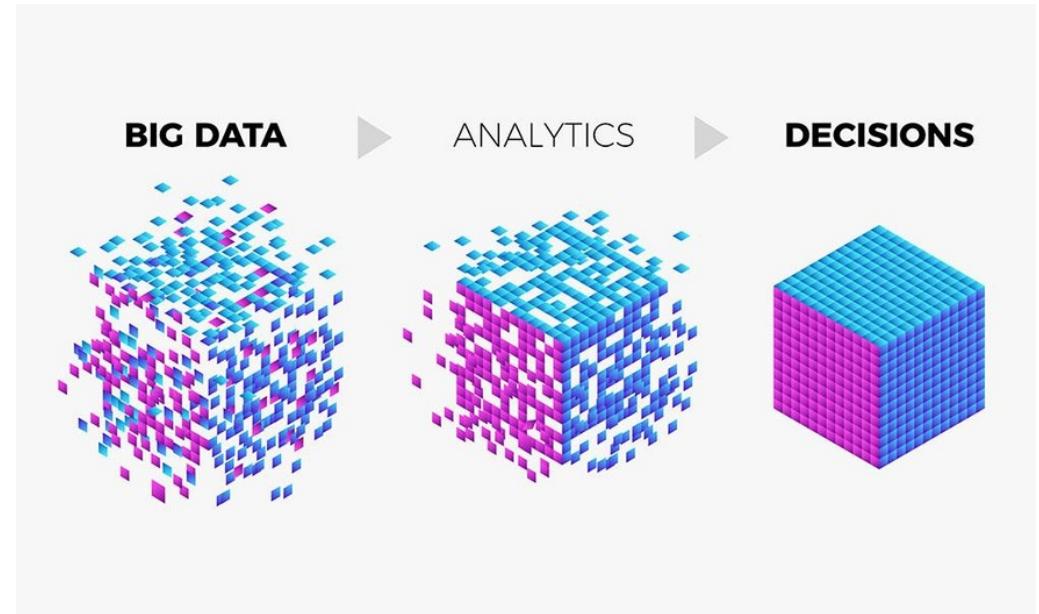
- Adobe Flex Builder
 - Flare (<http://flare.prefuse.org/>)
- Microsoft Silverlight
- JavaFX

Second generation RIA data visualisation

- HTML 5 + JavaScript + SVG + CSS
 - Client-side rendering
 - No plug-in is required
 - Mobile computing enabled
- D3.js (<https://d3js.org/>) - Data Driven Document

Methodological Challenges

- Lack of analysis functions.
- Not reproducible.
- Not extendable.
- Require to learn multiple technologies and methods.



Getting to Know Shiny

Shiny: Overview

- Shiny is an open package from RStudio.
- It provides a **web application framework** to create interactive web applications (visualization) called "Shiny apps".
- It can be found at
<https://shiny.rstudio.com/>



Getting to Know Shiny

What is so special about Shiny?

It allows R users:

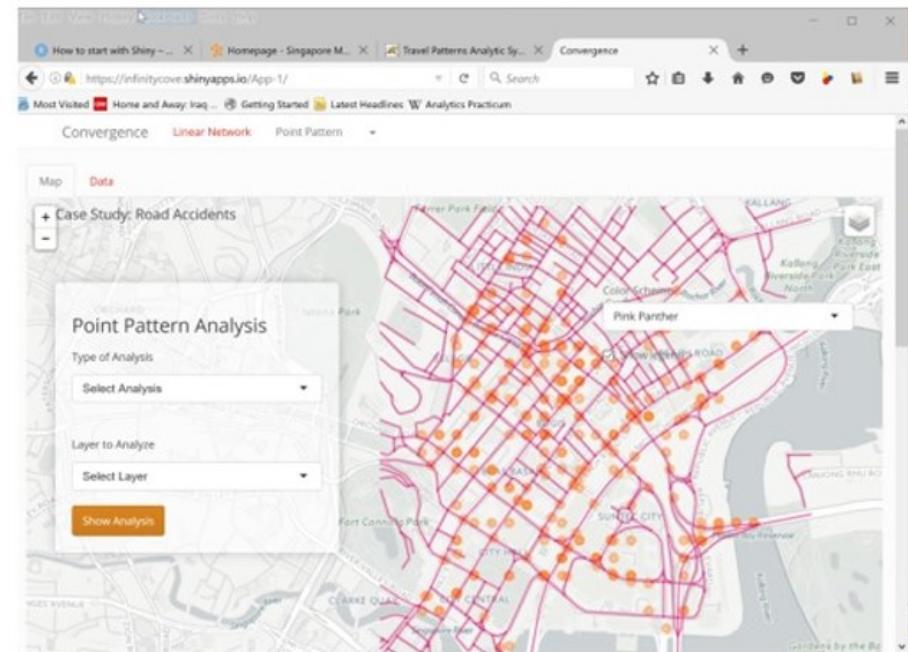
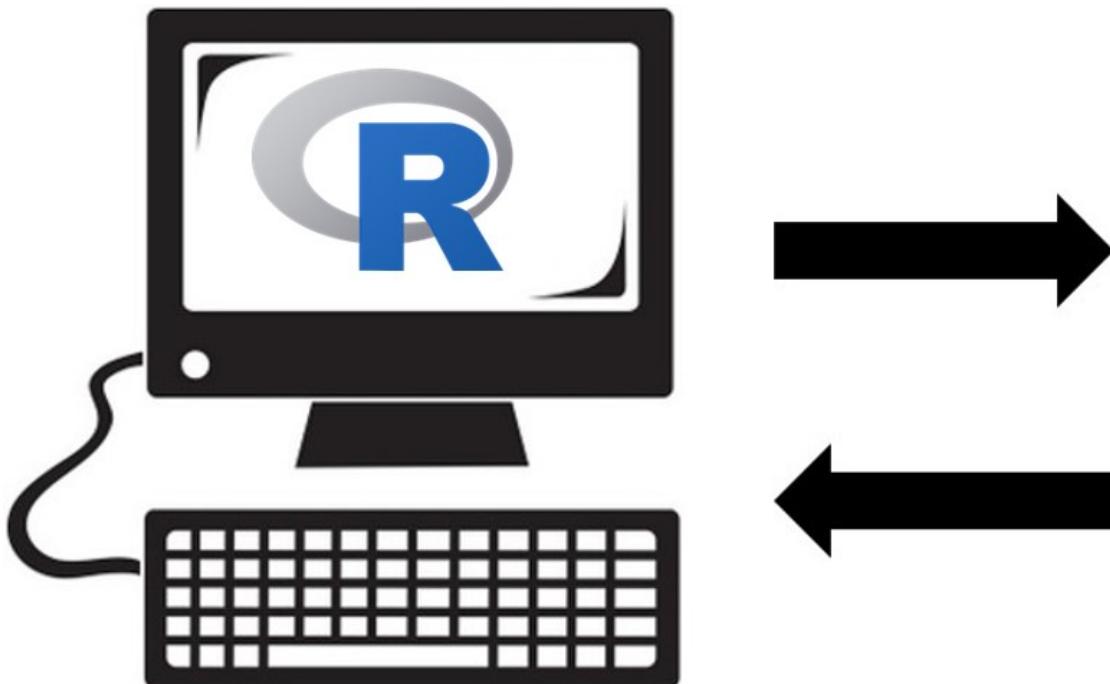
- to build and share highly interactive web-enabled applications without having to invest significant amount of time and efforts to master core web design technologies such as html5, Javascript and CSS.
- to integrate the analytical and visualisation packages of R without having to change from one programming language to another.



Getting to Know Shiny

Understanding the architecture

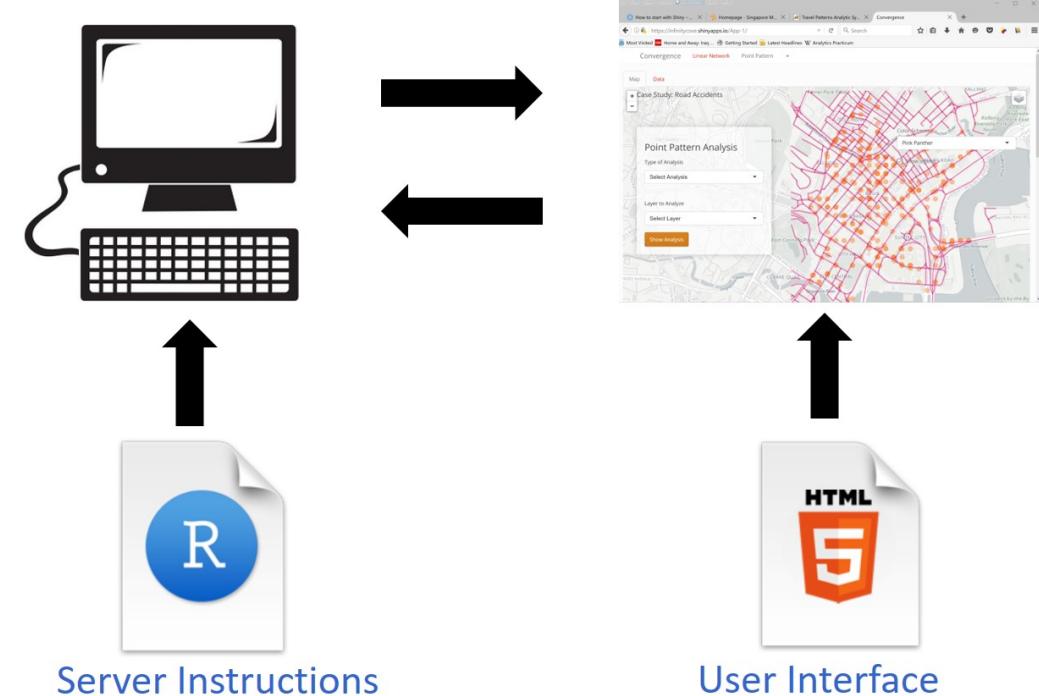
- Every Shiny app is maintained by a computer running R.



Getting to Know Shiny

The Structure of a Shiny app

- A Shiny app comprises of two components, namely:
 - a user-interface script, and
 - a server script.



Getting to Know Shiny

Shiny's user-interface, *ui.R*

- The *ui.R* script controls the layout and appearance of a shiny app.
 - It is defined in a source script name *ui.R*.
 - Actually, *ui* is a web document that the user gets to see, it is based on the famous Twitter bootstrap framework, which makes the look and layout highly customizable and fully responsive.
 - In fact, you only need to know R and how to use the shiny package to build a pretty web application. Also, a little knowledge of HTML, CSS, and JavaScript may help.

Getting to Know Shiny

Shiny's server *server.R*

- The *server.R* script contains the instructions that your computer needs to build your Shiny app.
- You are expected to:
 - know how to programme with R.
 - familiar with Tidyverse, specifically dplyr, tidyr and ggplot2

Getting to Know Shiny

Shiny Examples

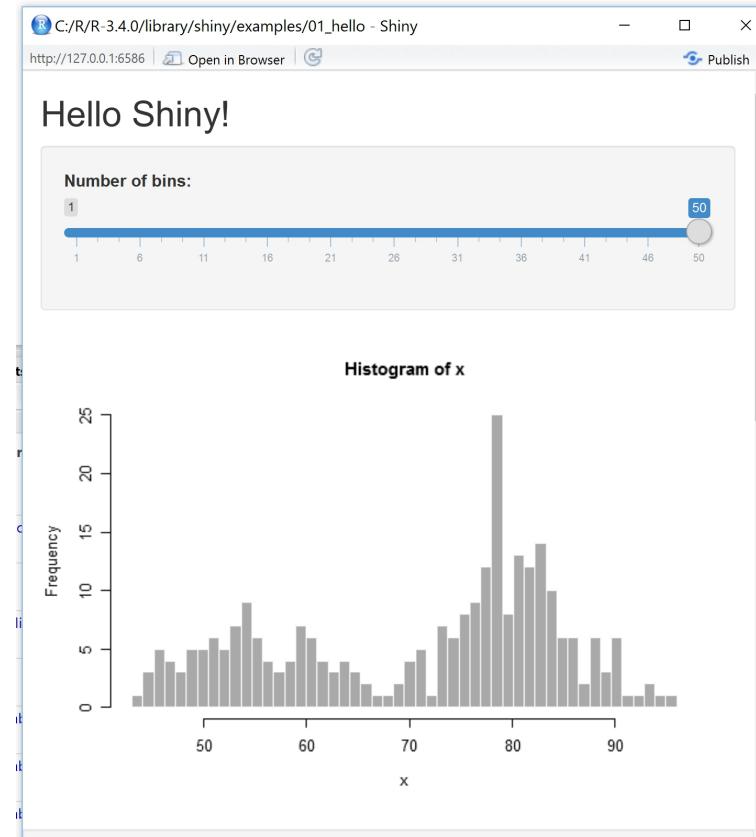
The Shiny package has eleven built-in examples that each demonstrates how Shiny works.

```
runExample("01_hello") # a histogram  
runExample("02_text") # tables and data frames  
runExample("03_reactivity") # a reactive expression  
runExample("04_mpg") # global variables  
runExample("05_sliders") # slider bars  
runExample("06_tabssets") # tabbed panels  
runExample("07_widgets") # help text and submit buttons  
runExample("08_html") # Shiny app built from HTML  
runExample("09_upload") # file upload wizard  
runExample("10_download") # file download wizard  
runExample("11_timer") # an automated timer
```

Getting to Know Shiny

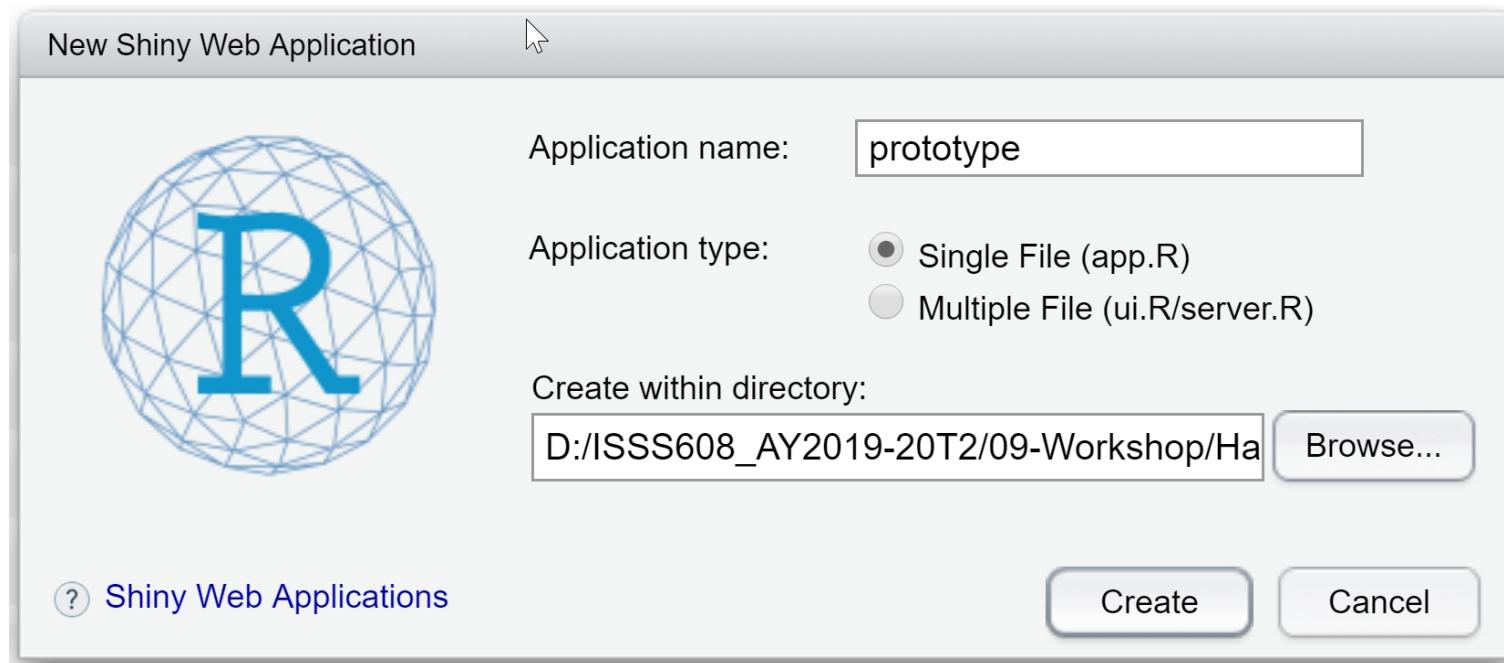
Runing Shiny example

```
library(shiny)  
runExample("01_hello")
```



Building a Shiny app

- A Shiny app can be in a form of a single file called *app.R*.
- Alternatively, a Shiny app can be also created using separate *ui.R* and *server.R* files.
- The separate files way is preferred when the app is complex and involves more codes.



Building a Shiny app

Survival Tips

- Always run the entire script, not just up to the point where you're developing code.
- Sometimes the best way to see what's wrong is to run the app and review the error.
- Watch out for commas!

Building a Shiny app

A basic Shiny app script

```
library(shiny)  
ui <- fluidPage()
```

User interface

controls the layout and appearance of app

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

Server function

contains instructions needed to build app

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

shinyApp()

Creates the Shiny app object

Building a Shiny app

Important tips of Shiny app file

- It is very important that the name of the file is *app.R*, otherwise it would not be recognized as a Shiny app.
- You should not have any R code after the `shinyApp(ui = ui, server = server)` line. That line needs to be the last line in your file.
- It is good practice to place this app in its own folder, and not in a folder that already has other R scripts or files, unless those other files are used by your app.

Loading the dataset

```
library(shiny)
library(tidyverse)

exam <- read_csv("data/Exam_data.csv")

ui <- fluidPage()
server <- function(input, output) {}
shinyApp (ui=ui, server=server)
```

- Make sure that the data file path and file name are correct.
- To check if the dataset has been added correctly, you can add a *print()* argument after reading the data.

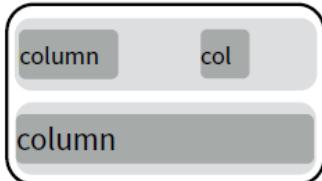
Shiny Layout

- Shiny ui.R scripts use the function *fluidPage* to create a display that automatically adjusts to the dimensions of your user's browser window.
- You lay out your app by placing elements in the *fluidPage* function.
- *titlePanel* and *sidebarLayout* are the two most popular elements to add to *fluidPage*. They create a basic Shiny app with a sidebar.

Shiny Layout Panels

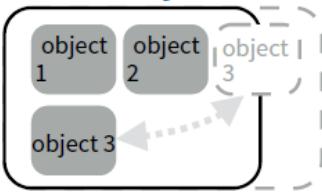
An overview

fluidRow()



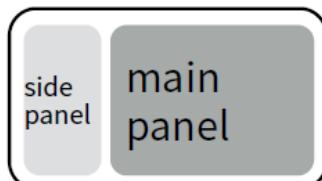
```
ui <- fluidPage(  
  fluidRow(column(width = 4),  
            column(width = 2, offset = 3)),  
  fluidRow(column(width = 12))  
)
```

flowLayout()



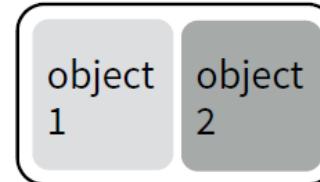
```
ui <- fluidPage(  
  flowLayout(# object 1,  
             # object 2,  
             # object 3)  
)
```

sidebarLayout()



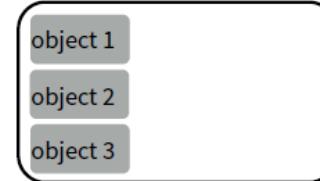
```
ui <- fluidPage(  
  sidebarLayout(  
    sidebarPanel(),  
    mainPanel())  
)
```

splitLayout()



```
ui <- fluidPage(  
  splitLayout(# object 1,  
             # object 2)  
)
```

verticalLayout()



```
ui <- fluidPage(  
  verticalLayout(# object 1,  
                 # object 2,  
                 # object 3)  
)
```

Shiny Layout Panels

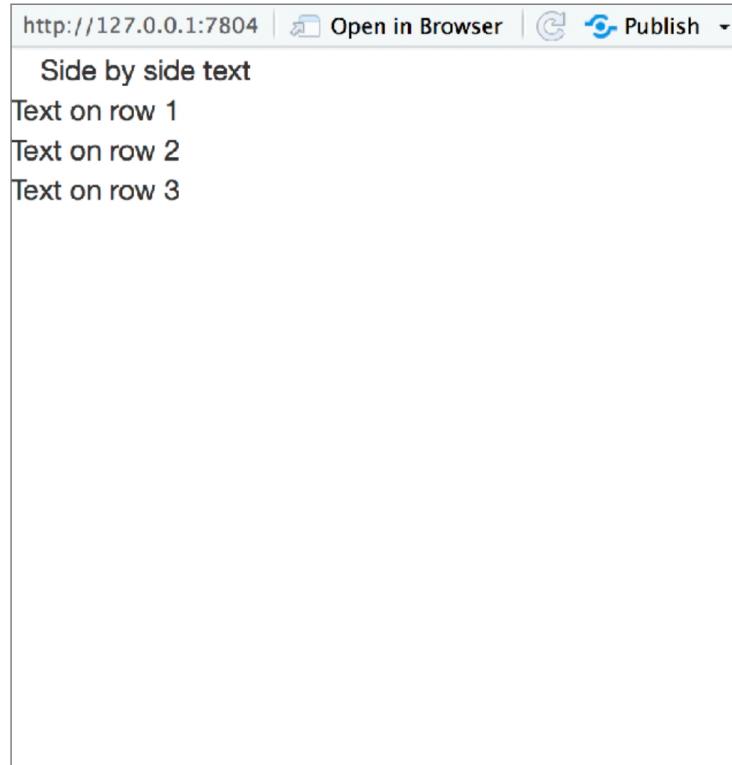
fluidrow()

```
library(shiny)

# Define UI with fluid rows
ui <- fluidPage(
  "Side", "by", "side", "text",
  fluidRow("Text on row 1"),
  fluidRow("Text on row 2"),
  fluidRow("Text on row 3")
)

# Define server fn that does nothing :)
server <- function(input, output) {}

# Create the app object
shinyApp(ui = ui, server = server)
```



Shiny Layout Panels

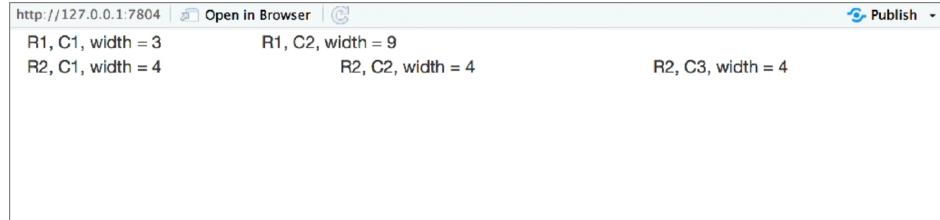
column()

```
library(shiny)

# Define UI with fluid rows and columns
ui <- fluidPage(
  fluidRow(
    column("R1", C1, width = 3, width = 3),
    column("R1", C2, width = 9, width = 9)
  ),
  fluidRow(
    column("R2", C1, width = 4, width = 4),
    column("R2", C2, width = 4, width = 4),
    column("R2", C3, width = 4, width = 4)
  )
)

# Define server fn that does nothing :)
server <- function(input, output) {}

# Create the app object
shinyApp(ui = ui, server = server)
```



Shiny Layout Panels

More about column()

The image displays two screenshots of a Shiny application running at <http://127.0.0.1:7804>. The top screenshot shows a layout with three columns in the first row: R1, C1, width = 3; R1, C2, width = 9; and R2, C3, width = 4. The bottom screenshot shows a layout with three columns in the first row: R1, C1, width = 3; R2, C1, width = 4; R1, C2, width = 9; R2, C2, width = 4; and R2, C3, width = 4. Orange arrows at the bottom of each screenshot indicate the horizontal span of the columns. Handwritten orange annotations on the right side of the top screenshot show the calculations $3 + 9 = 12$ and $4 + 4 + 4 = 12$.

http://127.0.0.1:7804 | Open in Browser | Publish

R1, C1, width = 3 R1, C2, width = 9 R2, C3, width = 4

R2, C1, width = 4 R2, C2, width = 4

3 + 9 = 12

4 + 4 + 4 = 12

http://127.0.0.1:7804 | Open in Browser | Publish

R1, C1, width = 3 R1, C2, width = 9 R2, C3, width = 4

R2, C1, width = 4 R2, C2, width = 4

Shiny Layout Panels

Panels

- Use panels to group multiple elements into a single element that has its own properties.
- Especially important and useful for complex apps with a large number of inputs and outputs such that it might not be clear to the user where to get started.

Shiny Layout Panels

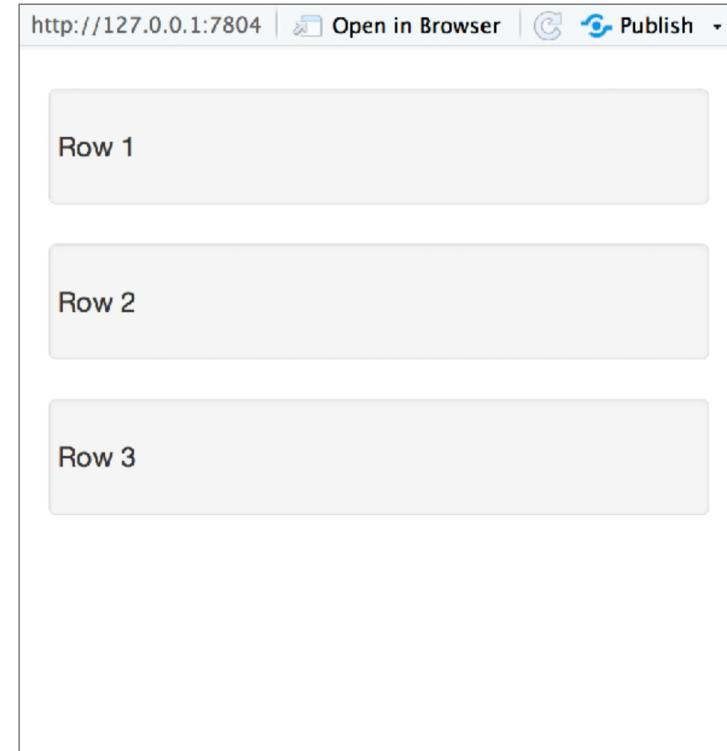
wellPanel()

```
library(shiny)

# Define UI with wellPanels
ui <- fluidPage(
  wellPanel( fluidRow("Row 1") ),
  wellPanel( fluidRow("Row 2") ),
  wellPanel( fluidRow("Row 3") )
)

# Define server fn that does nothing :)
server <- function(input, output) {}

# Create the app object
shinyApp(ui = ui, server = server)
```



Shiny Layout Panels

sidebarPanel() and mainPanel()

```
library(shiny)

# Define UI with default width sidebar
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel("Usually inputs go here"),
    mainPanel("Usually outputs go here")
  )
)

# Define server fn that does nothing :)
server <- function(input, output) {}

# Create the app object
shinyApp(ui = ui, server = server)
```



Shiny Layout Panels

Customising sidebarPanel() and mainPanel()

```
library(shiny)

# Define UI with custom width sidebar
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel("Usually inputs go here",
                width = 6),
    mainPanel("Usually outputs go here",
              width = 6)
  )
)

# Define server fn that does nothing :)
server <- function(input, output) {}

# Create the app object
shinyApp(ui = ui, server = server)
```



Shiny Layout Panels

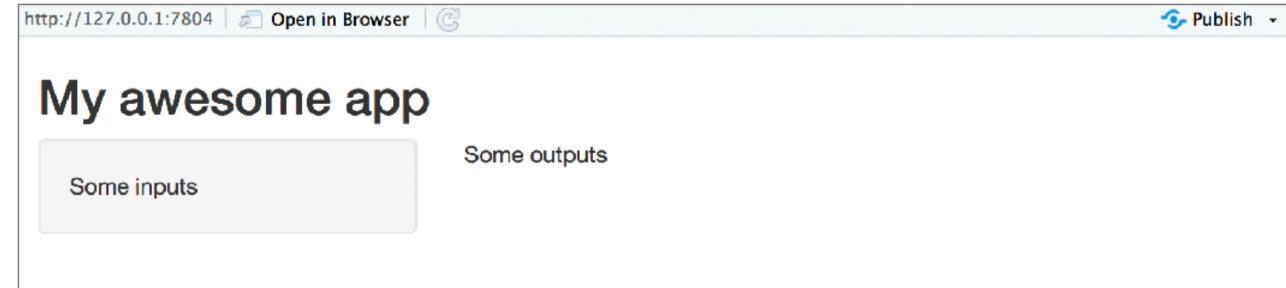
titlePanel()

```
library(shiny)

# Define UI with title panel
ui <- fluidPage(
  titlePanel("My awesome app"),
  sidebarLayout(
    sidebarPanel("Some inputs"),
    mainPanel("Some outputs")
  )
)

# Define server fn that does nothing :)
server <- function(input, output) {}

# Create the app object
shinyApp(ui = ui, server = server)
```



Building the basic UI

Working with *titlePanel*

- *titlePanel* is used to add the application title.

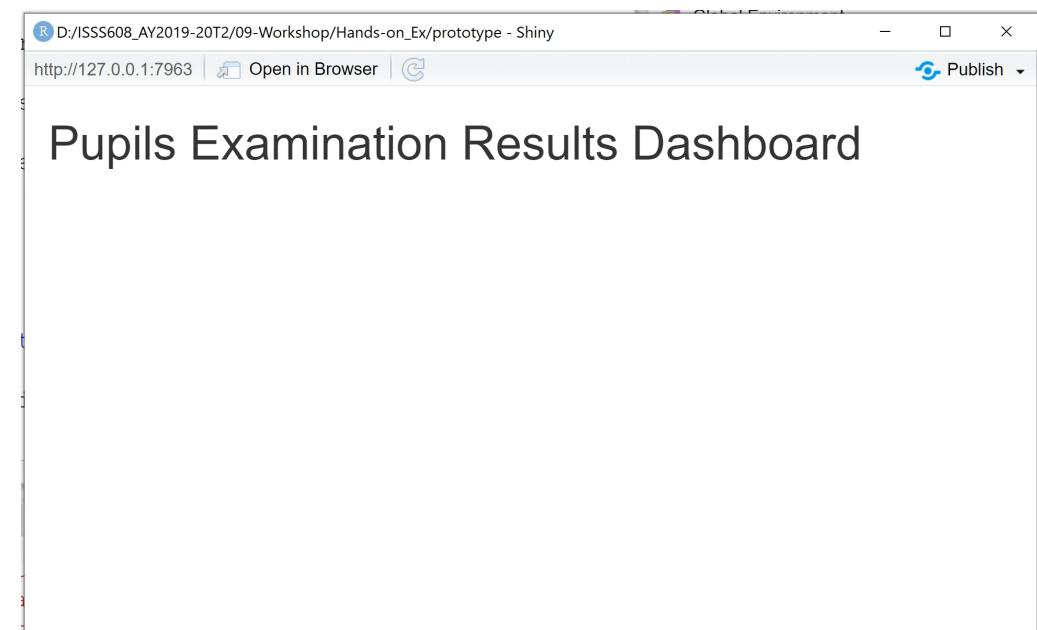
```
library(shiny)
library(tidyverse)

exam <- read_csv("data/Exam_data.csv")

ui <- fluidPage(
  titlePanel("Pupils Examination Results Dashboard")
)

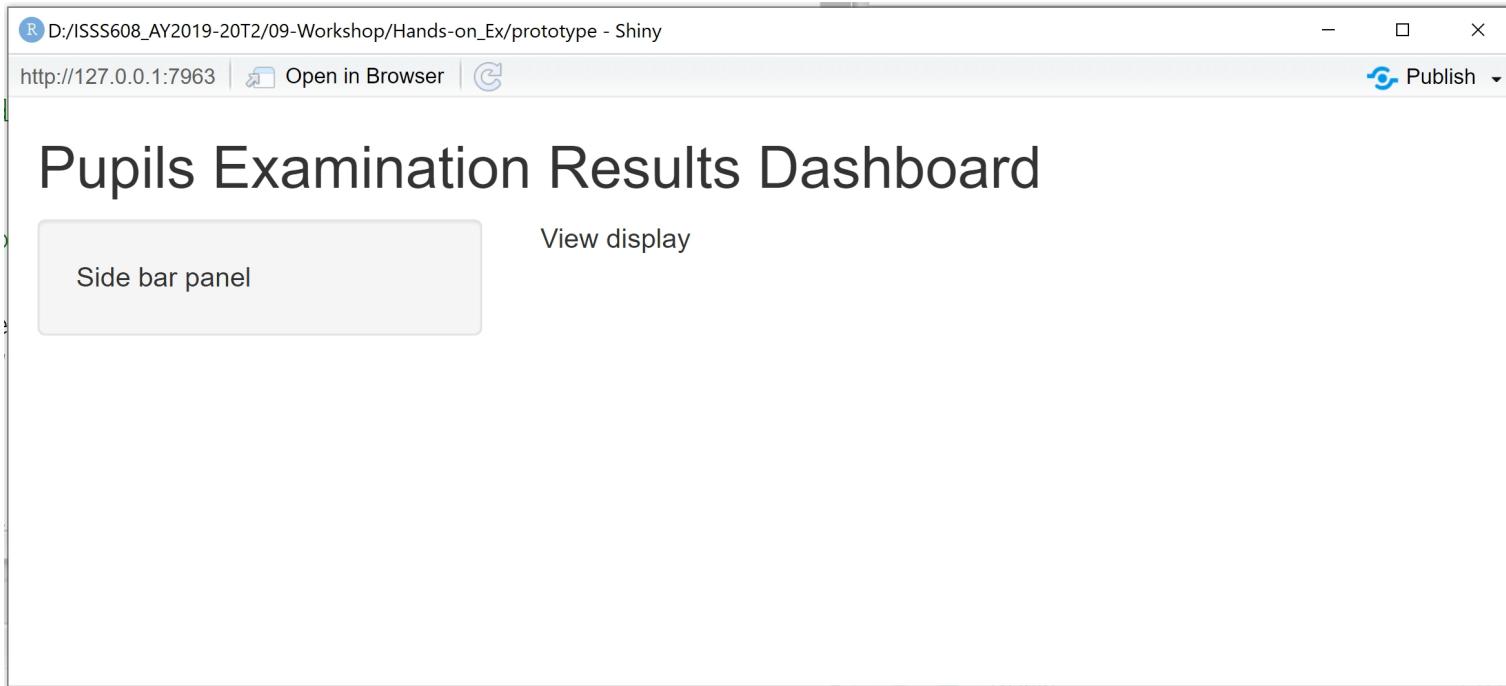
server <- function(input, output) {}

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



Building the basic UI

Working with *sidebarLayout()*



Building the basic UI

Working with *sidebarLayout()*

- *sidebarLayout()* always takes two arguments:
 - *sidebarPanel()* function output
 - *mainPanel()* function output
- These functions place content in either the sidebar or the main panels.
- The *sidebarPanel()* will appear on the left side of your app by default. You can move it to the right side by giving *sidebarLayout()* the optional argument `position = "right"`.
- You can use *navbarPage* to give your app a multi-page user-interface that includes a navigation bar. Or you can use *fluidRow()* and *column()* to build your layout up from a grid system.
- To learn more about the advanced options, read the Shiny Application Layout Guide [<http://shiny.rstudio.com/articles/layout-guide.html>].

Building the basic UI

Working with *sidebarLayout()*

```
library(shiny)
library(tidyverse)

exam <- read_csv("data/Exam_data.csv")

ui <- fluidPage(
  titlePanel("Pupils Examination Results Dashboard"),
  sidebarLayout(
    sidebarPanel("Side bar panel"),
    mainPanel("View display")
  )
)

server <- function(input, output) {}
shinyApp(ui=ui, server=server)
```

- Note that in a fluid design your sidebar and other elements may "collapse" if your browser view is not wide enough.

Shiny Inputs

An overview of Shiny Inputs

- Inputs are what gives users a way to interact with a Shiny app.
- Shiny provides many input functions to support many kinds of interactions that the user could have with an app.

Inputs

collect values from the user

Access the current value of an input object with `input$<inputId>`. Input values are **reactive**.

Action

`actionButton(inputId, label, icon, ...)`

Link

`actionLink(inputId, label, icon, ...)`

Choice 1
 Choice 2
 Choice 3
 Check me



`checkboxGroupInput(inputId, label, choices, selected, inline)`

`checkboxInput(inputId, label, value)`

`dateInput(inputId, label, value, min, max, format, startview, weekstart, language)`

`dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)`

Choose File

`fileInput(inputId, label, multiple, accept)`

1

`numericInput(inputId, label, value, min, max, step)`

.....

`passwordInput(inputId, label, value)`

Choice A
 Choice B
 Choice C

Choice 1
Choice 2

`radioButtons(inputId, label, choices, selected, inline)`

0 5 10

`selectInput(inputId, label, choices, selected, multiple, selectize, width, size) (also selectizeInput())`

Apply Changes

`sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)`

Enter text

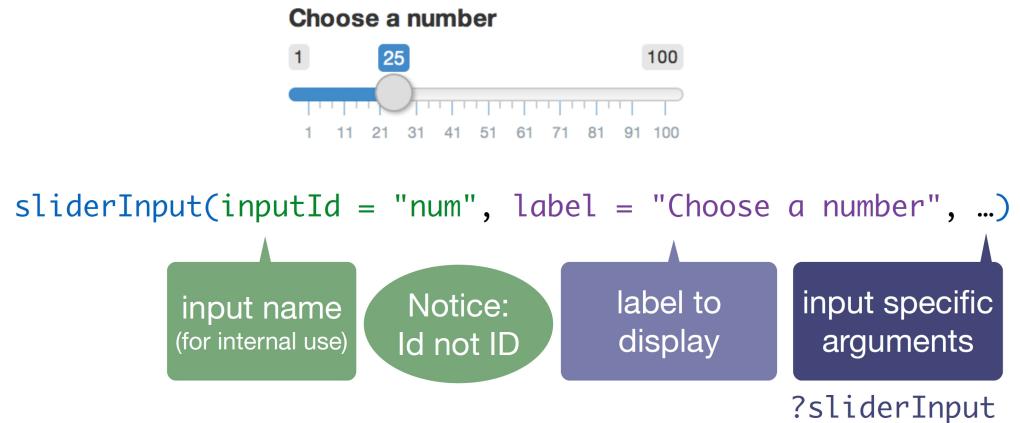
`submitButton(text, icon)`
(Prevents reactions across entire app)

`textInput(inputId, label, value)`

Shiny Inputs

Inputs syntax

- All input functions have the same first two arguments: `inputId` and `label`.
- The `inputId` will be the name that Shiny will use to refer to this input when you want to retrieve its current value.
- It is important to note that every input must have a unique `inputId`.
- The `label` argument specifies the text in the display label that goes along with the input widget.
- Every input can also have multiple other arguments specific to that input type.



Shiny Inputs

Adding inputs to the UI (*selectInput()* and *sliderInput()* functions)

```
ui <- fluidPage(  
  titlePanel("Pupils Examination Results Dashboard"),  
  sidebarLayout(  
    sidebarPanel(  
      selectInput(inputId = "variable",  
                  label = "Subject:",  
                  choices = c("English" = "ENGLISH",  
                             "Maths" = "MATHS",  
                             "Science" = "SCIENCE"),  
                  selected = "ENGLISH"),  
      sliderInput(inputId = "bins",  
                  label = "Number of Bins",  
                  min = 5,  
                  max = 20,  
                  value= 10)  
    ),  
    mainPanel()  
  )  
)
```

Shiny Inputs

Adding `checkboxInput()`

Add a checkbox input to specify whether the data plotted should be shown in a data table.

- ui: Add an input widget that the user can interact with to check/uncheck the box.
- ui: Add an output defining where the data table should appear.
- server: Add a reactive expression that creates the data table if the checkbox is checked.

Shiny Inputs

Adding *checkboxInput()*

- ui: Add an input widget that the user can interact with to check/uncheck the box.

```
sliderInput(inputId = "bin",
            label = "Number of Bins",
            min = 5,
            max = 20,
            value = c(10)),
checkboxInput(inputId = "show_data",
              label = "Show data table",
              value = TRUE)
),
mainPanel()
)
)
```

Shiny *Output()*

An overview of Shiny *Output()*

- After creating all the inputs, we should add elements to the UI to display the outputs.
 - To display output, add it to `fluidPage()` with an `Output()` function.

```
plotOutput("hist")
```

the type of output
to display

name to give to the output object

© CC 2015 RStudio,

- Similarly to the input functions, all the output functions have a `outputId` argument that is used to identify each output, and this argument must be unique for each output.
 - Each output needs to be constructed in the server code later.

Shiny Output()

Shiny Output() options

- Outputs can be any object that R creates and that we want to display in our app - such as a plot, a table, or text.

Function	Inserts
dataTableOutput()	an interactive table
htmlOutput()	raw HTML
imageOutput()	image
plotOutput()	plot
tableOutput()	table
textOutput()	text
uiOutput()	a Shiny UI element
verbatimTextOutput()	text

Shiny *Output()*

Adding *plotOutput()*

```
ui <- fluidPage(  
  titlePanel("Pupils Examination Results Dashboard"),  
  sidebarLayout(  
    sidebarPanel("Side bar panel"),  
    mainPanel(  
      plotOutput("distPlot")  
    )  
  )  
)
```

Shiny *Output()*

Adding *dataTableOutput()*

- ui: Add an output defining where the data table should appear.

```
mainPanel(  
  plotOutput("distPlot"),  
  DT::dataTableOutput(outputId = "examtable")  
)
```

Shiny server.R

Building an output

There are three rules to build an output in Shiny, they are:

- Save the output object into the output list (remember the app template - every server function has an output argument).
- Build the object with a *render()* function, where is the type of output.
- Access input values using the input list (every server function has an input argument)

Note: The third rule is only required if you want your output to depend on some input.

Shiny server.R

A generic Shiny *Render()* syntax

```
renderPlot({ hist(rnorm(100)) })
```

type of object to build

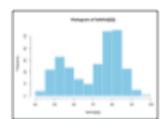
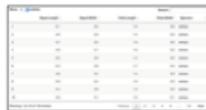
code block that builds the object

Shiny server.R

Shiny *Render()*

Outputs -

render*() and *Output() functions work together to add R output to the UI



```
'data.frame': 3 obs. of  2 variables:  
 $ Sepal.Length: num  5.1 4.9 4.7  
 $ Sepal.Width : num  3.5 3 3.2
```

Sepal.Length	Sepal.Width
5.1	3.5
4.9	3.0
4.7	3.2

foo



DT::renderDataTable(expr, options, callback, escape, env, quoted)

renderImage(expr, env, quoted, deleteFile)

renderPlot(expr, width, height, res, ..., env, quoted, func)

renderPrint(expr, env, quoted, func, width)

renderTable(expr, ..., env, quoted, func)

renderText(expr, env, quoted, func)

renderUI(expr, env, quoted, func)

works
with

dataTableOutput(outputId, icon, ...)

imageOutput(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)

plotOutput(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)

verbatimTextOutput(outputId)

tableOutput(outputId)

textOutput(outputId, container, inline)

& **uiOutput(outputId, inline, container, ...)**

htmlOutput(outputId, inline, container, ...)

Shiny server.R

Building a basic output

Let's first see how to build a very basic output using only the first two rules. We'll create a plot and send it to the ***distPlot*** output.

```
server <- function(input, output){  
  output$distPlot <- renderPlot({  
    ggplot(exam, aes(ENGLISH)) +  
      geom_histogram(bins = 20,  
                      color="black",  
                      fill="light blue")  
  })  
}
```

- This simple code shows the first two rules: we're creating a plot inside the *renderPlot()* function, and assigning it to ***distPlot*** in the output list.
- Remember that every output created in the UI must have a unique ID, now we see why. In order to attach an R object to an output with ID x, we assign the R object to ***output\$distPlot***.
- Since ***distPlot*** was defined as a ***plotOutput***, we must use the *renderPlot()* function, and we must create a plot inside the *renderPlot()* function.

Shiny server.R

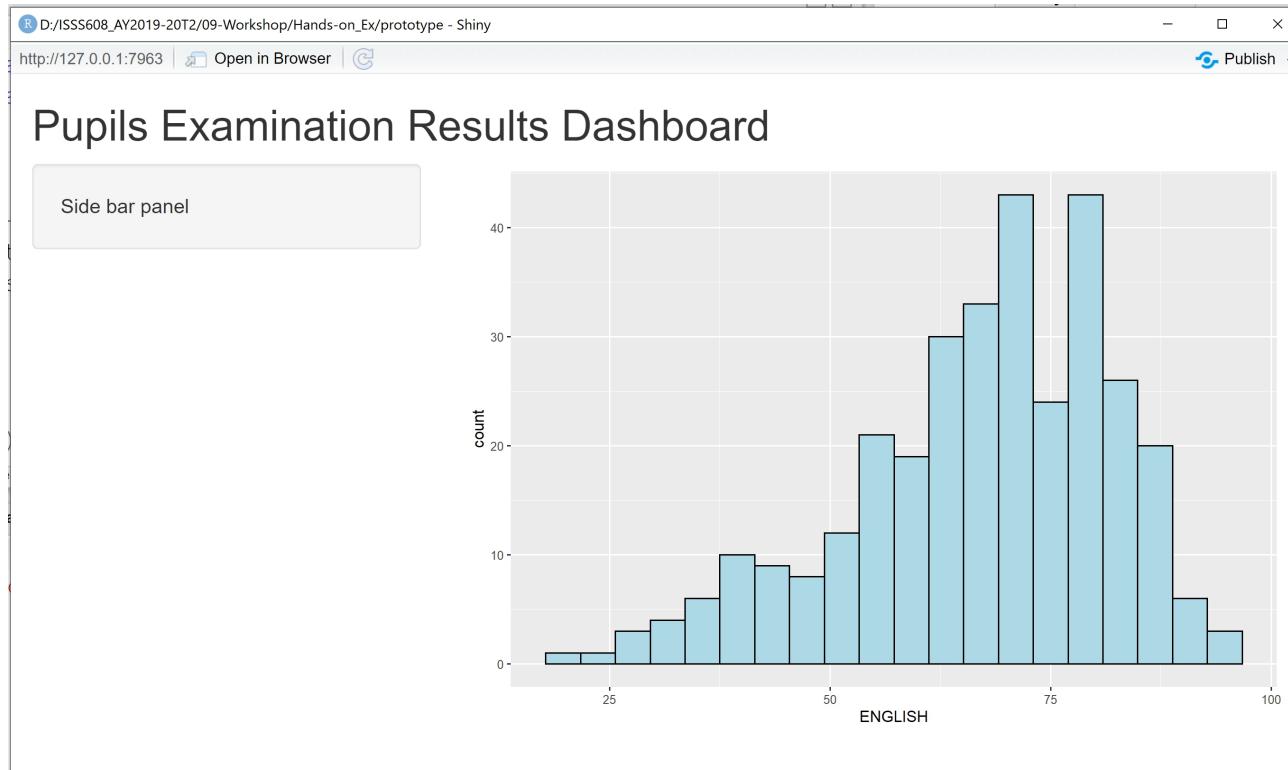
Building a complete output

```
server <- function(input, output){  
  output$distPlot <- renderPlot({  
    x <- unlist(exam[,input$variable])  
  
    ggplot(exam, aes(x)) +  
      geom_histogram(bins = input$bin,  
                     color="black",  
                     fill="light blue")  
  })  
  
  output$examtable <- DT:::renderDataTable({  
    if(input$show_data){  
      DT:::datatable(data = exam %>% select(1:7),  
                     options= list(pageLength = 10),  
                     rownames = FALSE)  
    }  
  })  
}
```

The shinyApp()

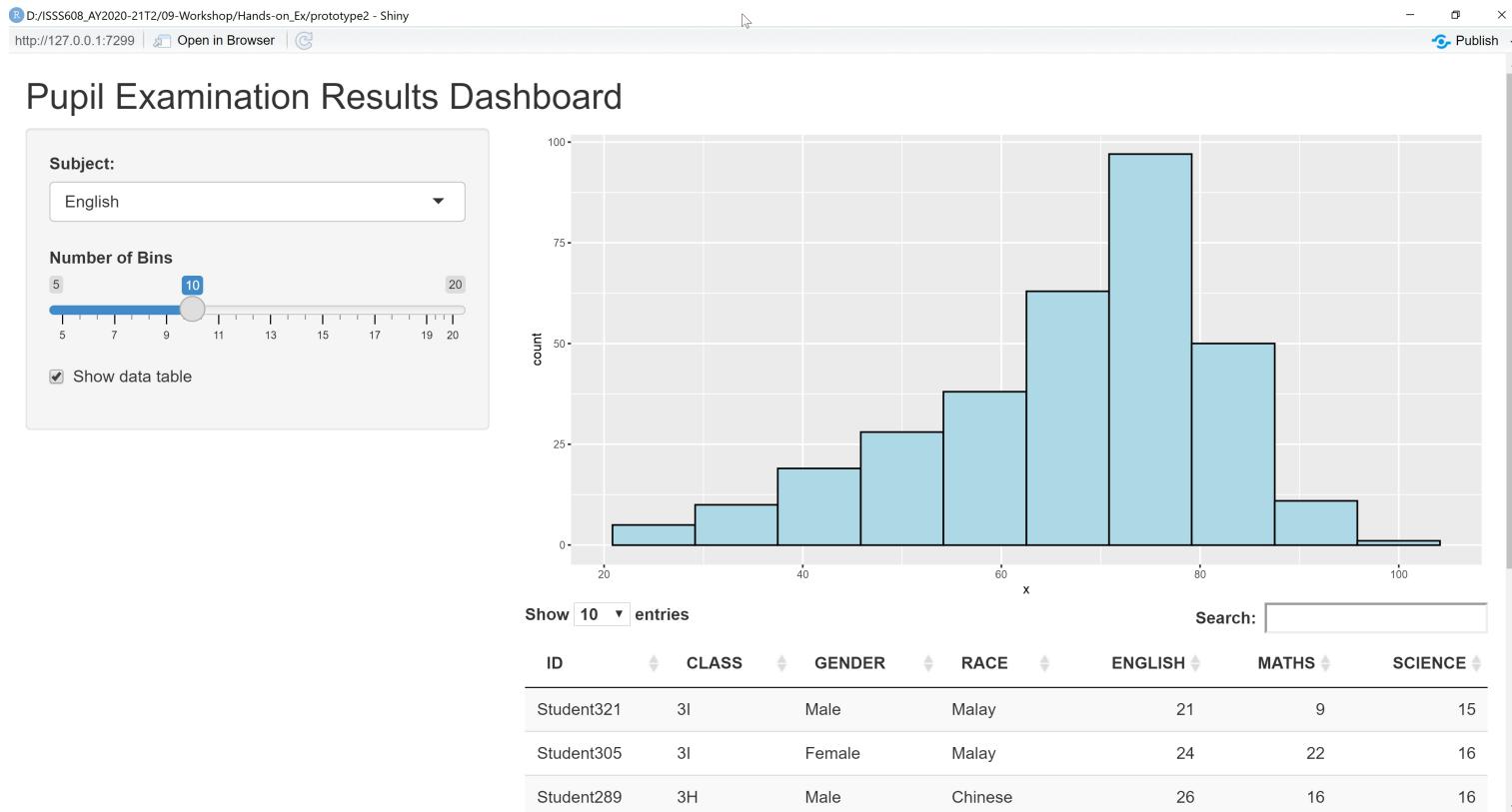
- It is important to add *shinyApp()* at the end of your Shiny application.

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

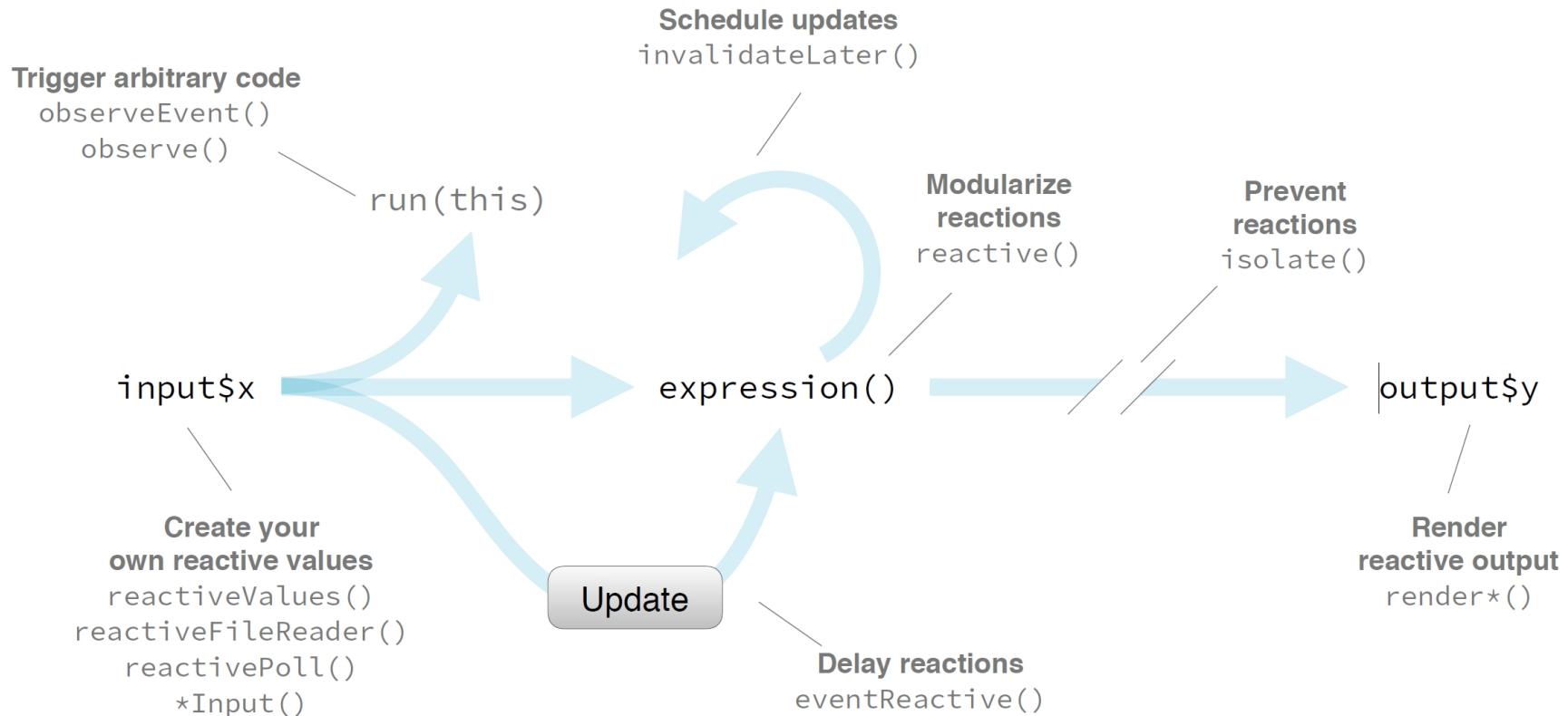


Shiny app is Reactive

- Reactivity automatically occurs when an input value is used to render an output object.

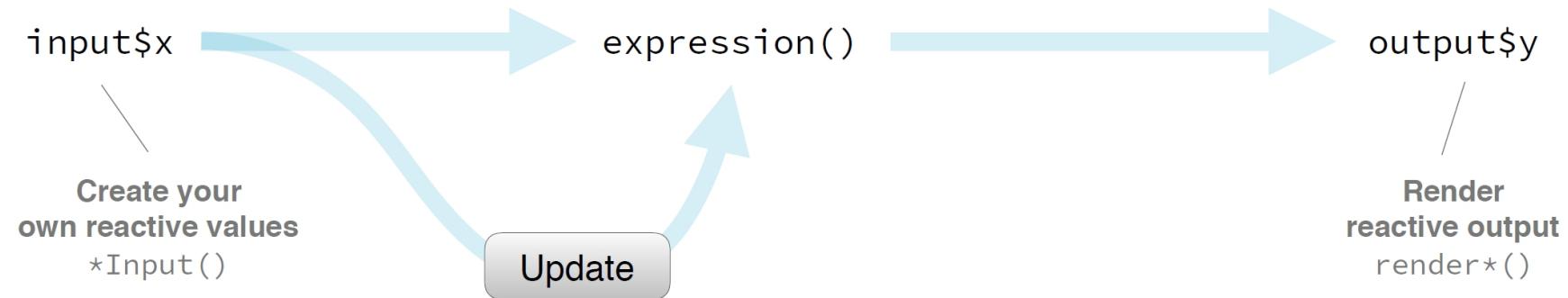


Reactive Flow



Reactive Flow

Reactive flow, simplified



References

- Hadley Wickham (2021) ['"Mastering Shiny"'](#), O'Reilly Media. This is a highly recommended book.
- [Building Web Applications with Shiny](#), especially Module 1 and 2.
- [Shiny Three Parts Tutorial](#).
- [Online Function reference](#)
- [The basic parts of a Shiny app](#)
- [How to build a Shiny app](#)
- [The Shiny Cheat sheet](#)

Beyond Uncle Google! Last but not least, when you need help

- [How to get help](#)