Building Web-enabled Visual Analytics Application with R Shiny: Beyond the basic

Dr. Kam Tin Seong
Assoc. Professor of Information Systems

School of Computing and Information Systems,
Singapore Management University

02 March 2022

Overview

In this lesson, selected advanced methods of Shiny will be discussed. You will also gain hands-on experiences on using these advanced methods to build Shiny applications.

By the end of this lesson, you will be able to:

- understanding the basic development cycle of creating apps, making changes, and experimenting with the results,
- debug errors in the codes,
- build complex Shiny application using module, and
- improve the productivity of Shiny applications development

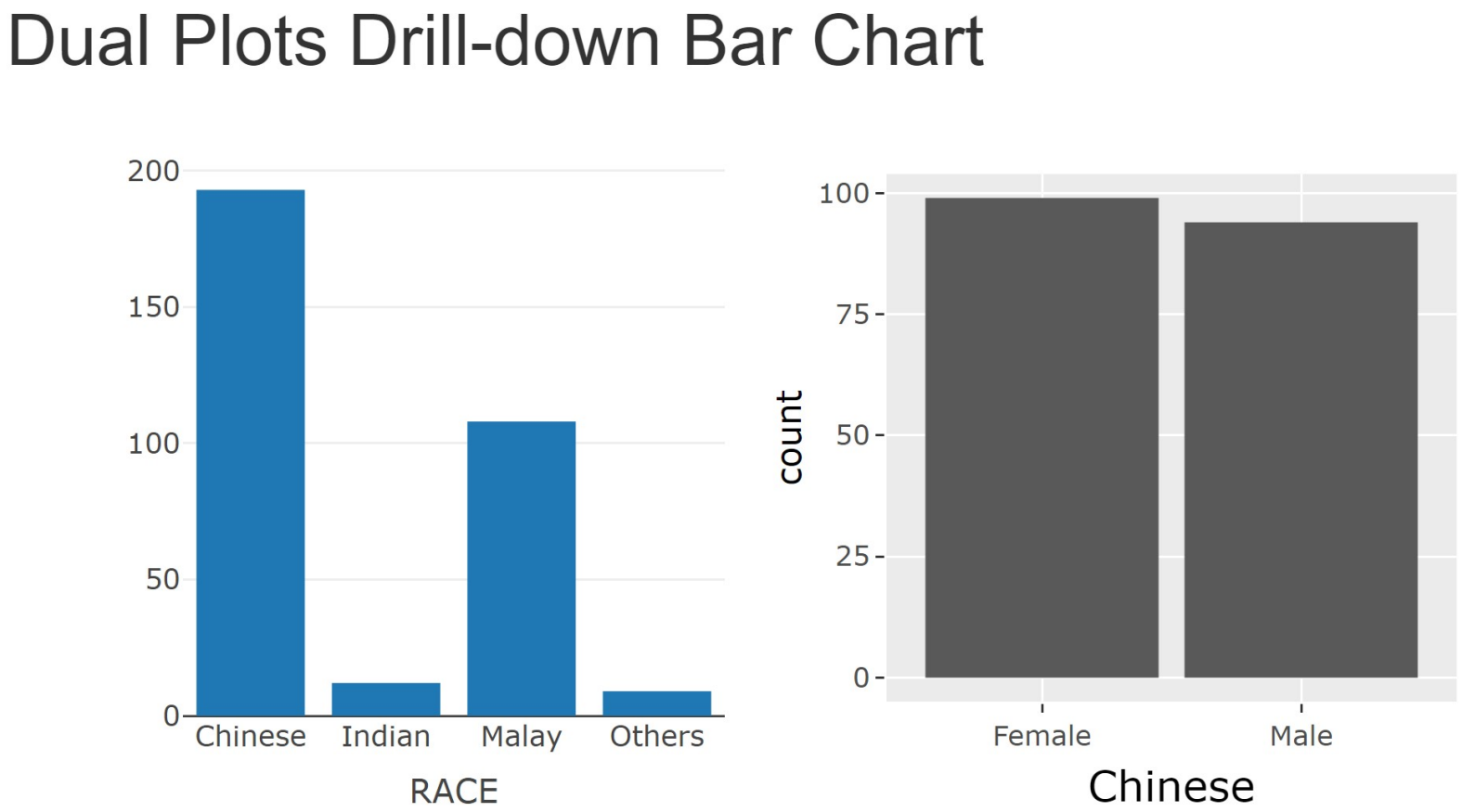
Working with Shiny Layout

Shiny includes a number of facilities for laying out the components of an application. This guide describes the following application layout features:

- A `sidebarLayout()`: for placing a `sidebarPanel()` of inputs alongside a `mainPanel()` output content.
- Custom layouts using Shiny's grid layout system (i.e., `fluidRow()` & `column()`).
- Segmenting layouts using the `tabsetPanel()` and `navlistPanel()` functions.
- Creating applications with multiple top-level components using the `navbarPage()` function.

Customising layout with `fluidRow()` and `column()`

In this hands-on exercise, you will learn how to rearrange the drill-down bar chart view so that both the Race and Gender bar charts will be placed next to each other as shown in the figure below.



Customising layout with `fluidRow()` and `column()`

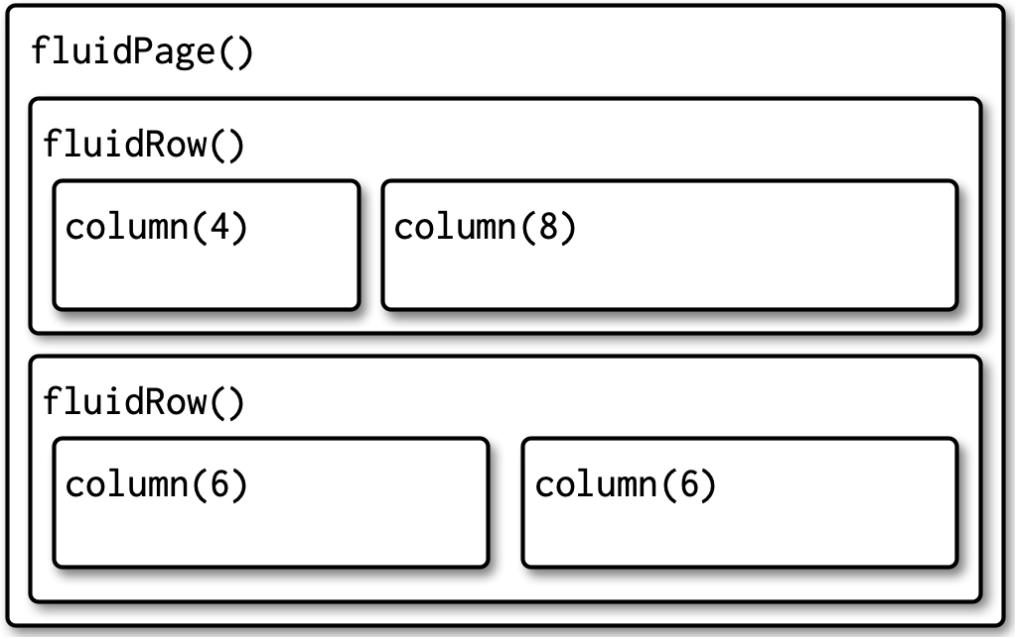
```
ui <- fluidPage(
  titlePanel("Dual Plots Drill-down Bar Chart"),
  mainPanel(
    fluidRow(
      column(6,
             plotlyOutput(
               outputId="race",
               width="300px",
               height="300px")),
      column(6,
             plotlyOutput(
               outputId="gender",
               width="300px",
               height="300px"))
    )
  )
)
```

What can you learn from the code chunk on the left?

- The first argument of the column function is width, which is based on the Bootstrap 12-wide grid system, so the total width for each row should add up to **12**.
- Shiny has numerous "Panel" functions, which you can think as the elements you put inside the layout containers you created using the "Layout" functions like `sidebarLayout()`, `fluidrow()`, `column()`, etc.

Customising Multi-row layout with `fluidRow()` and `column()`

`sidebarLayout()` is built on top of a flexible multi-row layout, which you can use directly to create more visually complex apps.



As usual, you start with `fluidPage()`. Then you create rows with `fluidRow()`, and columns with `column()`.

```
fluidPage(
  fluidRow(
    column(4,
      ...
    ),
    column(8,
      ...
    )
  ),
  fluidRow(
    column(6,
      ...
    ),
    column(6,
      ...
    )
  )
)
```

Multi-page layouts

Shiny provides several functions for building multi-page layout, they are:

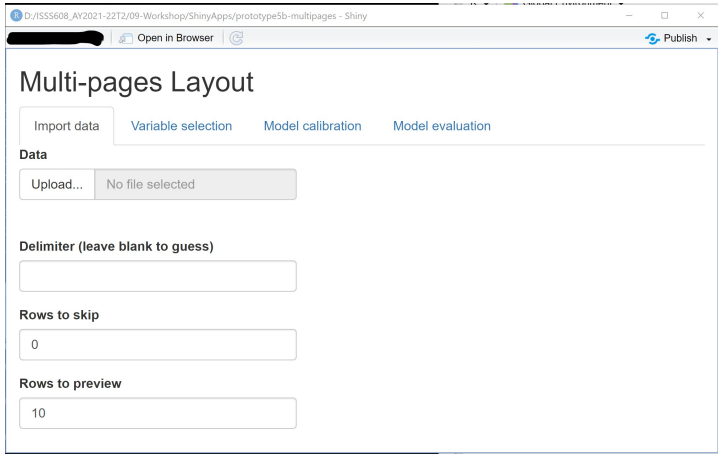
- tabsetPanel() + tabPanel()
- navlistPanel() + tabPanel()
- navbarMenu() + tabPanel()

tabsetPanel() and tabPanel()

The simple way to break up a page into pieces is to use `tabsetPanel()` and its close friend `tabPanel()`.

```
ui <- fluidPage(
  titlePanel("Multi-pages Layout"),
  tabsetPanel(
    tabPanel("Import data",
             fileInput("file", "Data",
                       buttonLabel = "Upload..
             textInput("delim",
                       "Delimiter (leave blank
                       ""),
             numericInput("skip", "Rows to skip
                          0, min = 0),
             numericInput("rows", "Rows to prev
                          10, min = 1)
    ),
    tabPanel("Set parameters"),
    tabPanel("Visualise results")
  )
)
```

Figure below shows the multi-page layouts created using the code chunk on the left.

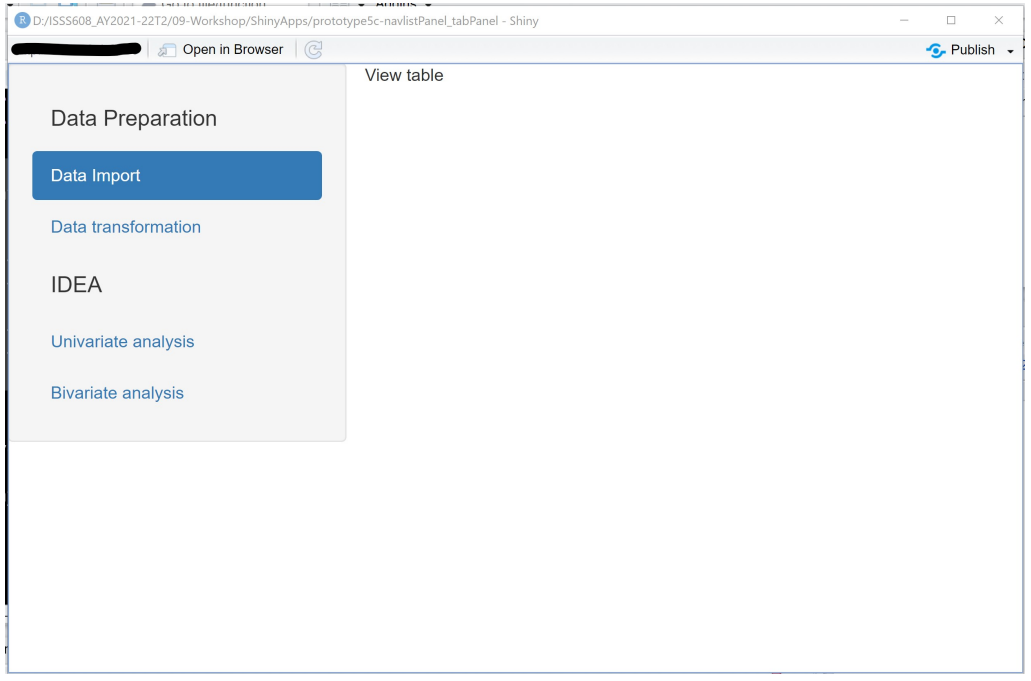


Note `tabsetPanel()` can be used anywhere in your app; it's totally fine to nest tabsets inside of other components (including tabsets!) if needed.

navlistPanel() and tabPanel()

- `navlistPanel()` is similar to `tabsetPanel()` but instead of running the tab titles horizontally, it shows them vertically in a sidebar.
- It also allows you to add headings with plain strings.

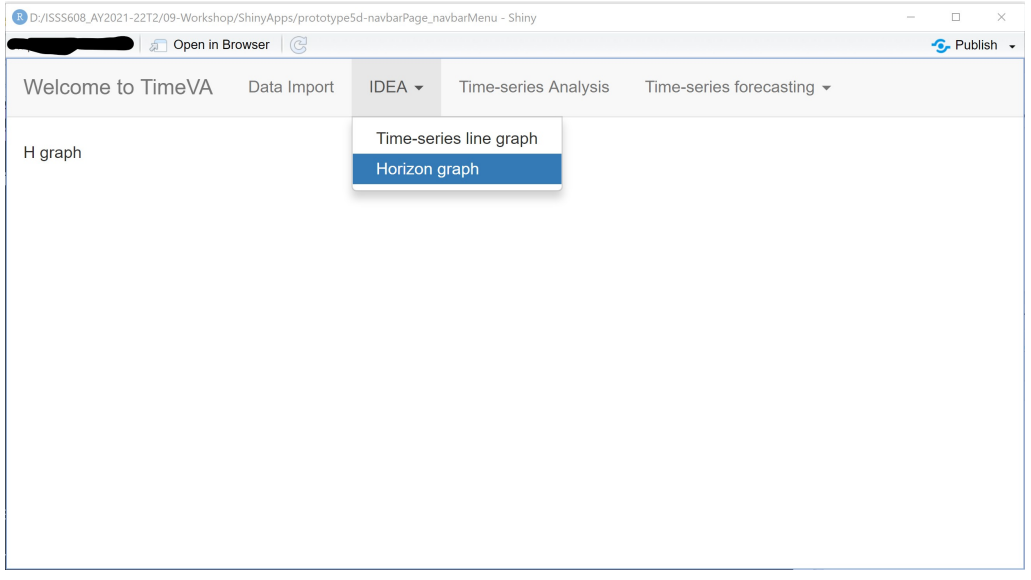
```
ui <- fluidPage(
  navlistPanel(
    id = "tabset",
    "Data Preparation",
    tabPanel("Data Import",
             "View table"),
    tabPanel("Data transformation",
             "Output table"),
    "IDEA",
    tabPanel("Univariate analysis",
             "Distribution plot"),
    tabPanel("Bivariate analysis",
             "Correlation matrix")
  )
)
```



navbarMenu() and tabPanel()

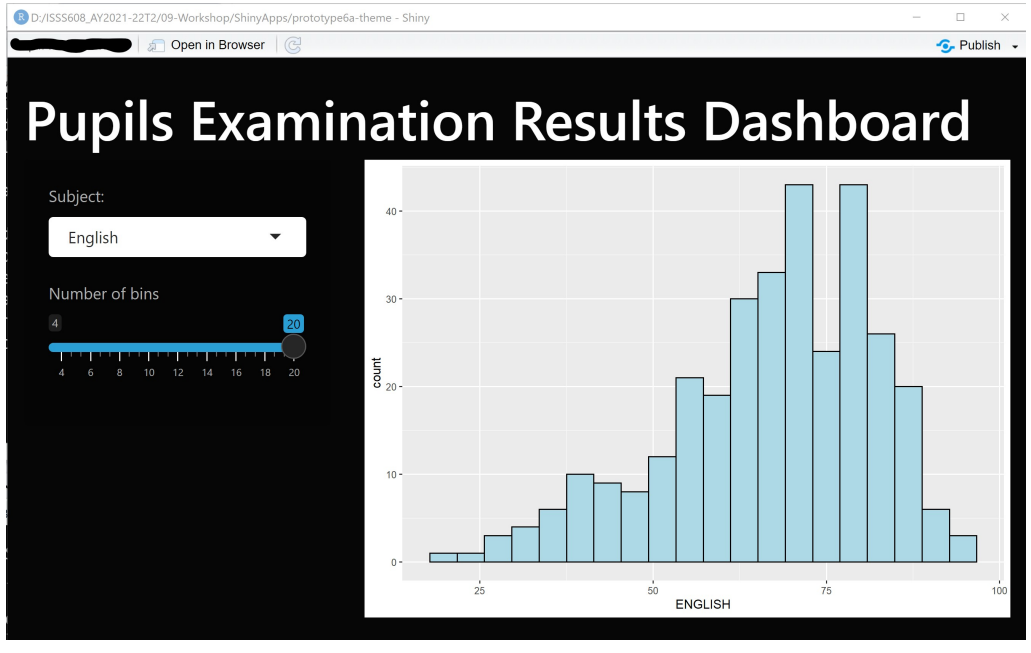
Another approach is use `navbarPage()`: it still runs the tab titles horizontally, but you can use `navbarMenu()` to add drop-down menus for an additional level of hierarchy.

```
ui <- navbarPage(
  "Welcome to TimeVA",
  tabPanel("Data Import", "one"),
  navbarMenu("IDEA",
             tabPanel("Time-series line graph"
             tabPanel("Horizon graph", "H grapl
             ),
  tabPanel("Time-series Analysis", "TSA"),
  navbarMenu("Time-series forecasting",
             tabPanel("Exponential Smoothing",
             tabPanel("ARIMA", "ARIMA"),
             tabPanel("Automatic", "auto")
  )
)
```



Shiny Themes

- Shiny v1.6 and higher integrates with the **bslib** package providing easy access to modern versions of Bootstrap, Bootswatch themes, as well as custom themes that can even be modified in real time!
- To use **bslib** in your own Shiny app, pass a `bs_theme()` object to the theme argument of the relevant page layout function, such as `navbarPage()` or `fluidPage()`.
- Inside `bs_theme()`, you can specify a version of Bootstrap and (optionally) a Bootswatch theme (e.g. cyborg)




```
library(shiny)
library(tidyverse)
library(bslib)
exam <- read_csv("data/Exam_data.csv")
ui <- fluidPage(
  theme = bs_theme(bootswatch = "cyborg"),
  titlePanel("Pupils Examination Results Dashbo
```

Custom themes

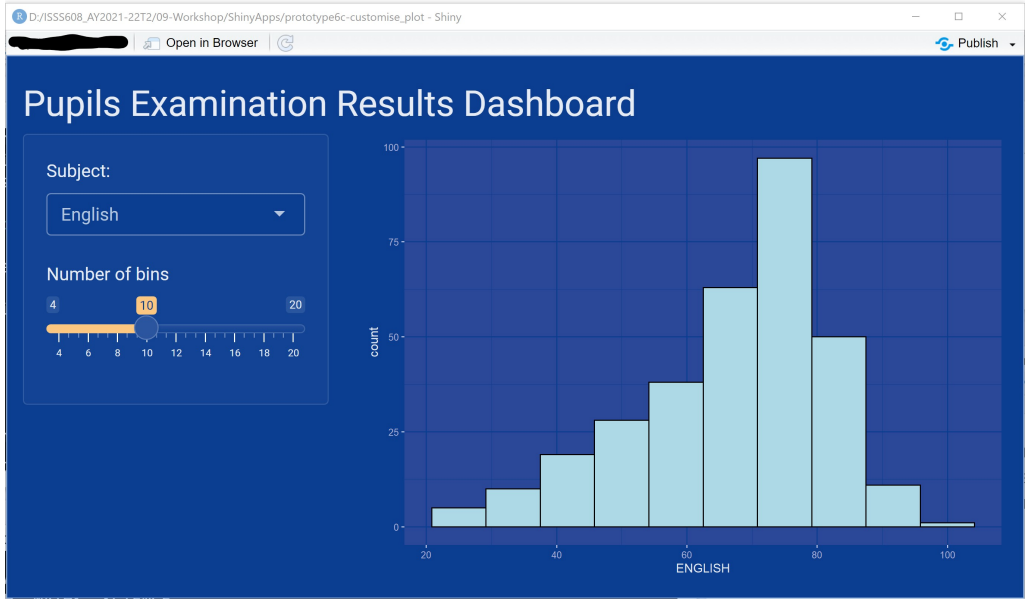
`bs_theme()` provides direct access to Bootstrap's main colors & fonts as well as any of the 100s of more specific theming options. Also, when it comes to custom font(s) that may not be available on the end users machine, make sure to leverage {bslib}'s helper functions like `font_google()`, `font_link()`, and `font_face()`, which assist in including font file(s) in an convenient, efficient, and responsible way.

```
library(shiny)
library(tidyverse)
library(bslib)

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

ui <- fluidPage(
  theme = bs_theme(bg = "#0b3d91",
                   fg = "white",
                   primary = "#FCC780",
                   base_font = font_google("Roboto"),
                   code_font = font_google("Roboto")),
  titlePanel("Pupils Examination Results Dashboard"),
```

Customising Plot

While a lot of custom theming can be done via `bs_theme()` (i.e., CSS), it fundamentally can't effect `renderPlot()`, because the image is rendered by R, not by the web browser. To help solve this problem, thematic package can be used to translate CSS to new R plotting defaults by just calling `thematic::thematic_shiny()` before running an app.

```
library(shiny)
library(tidyverse)
library(bslib)

thematic::thematic_shiny()

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

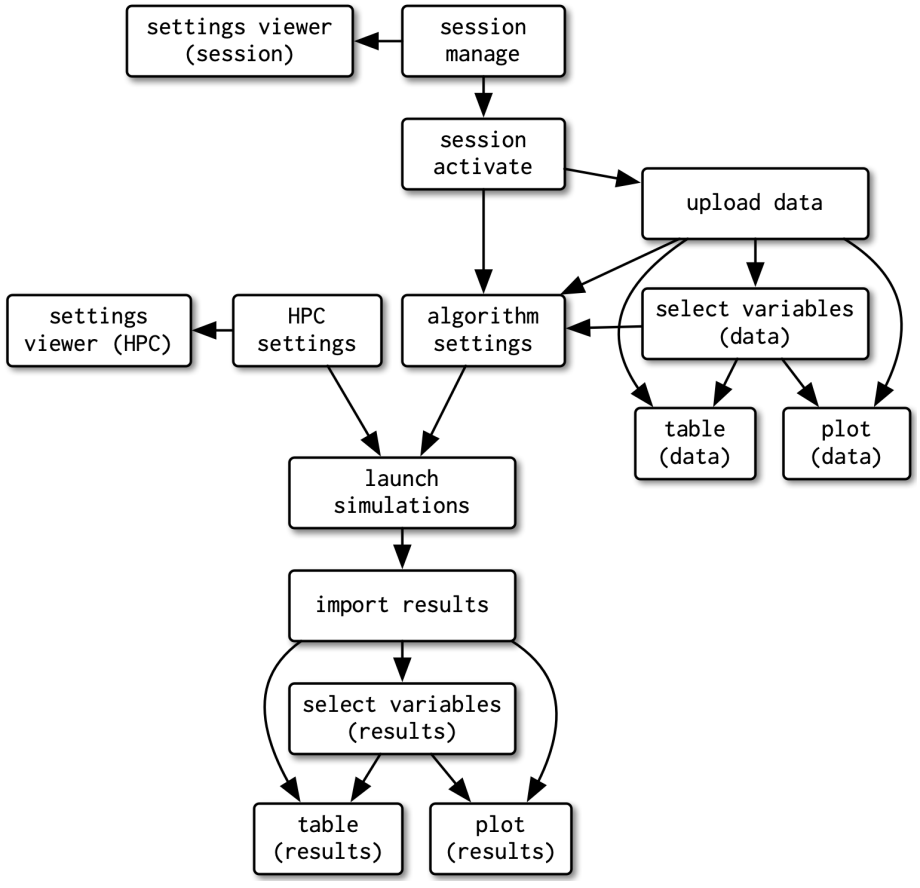


Introducing Shiny Module

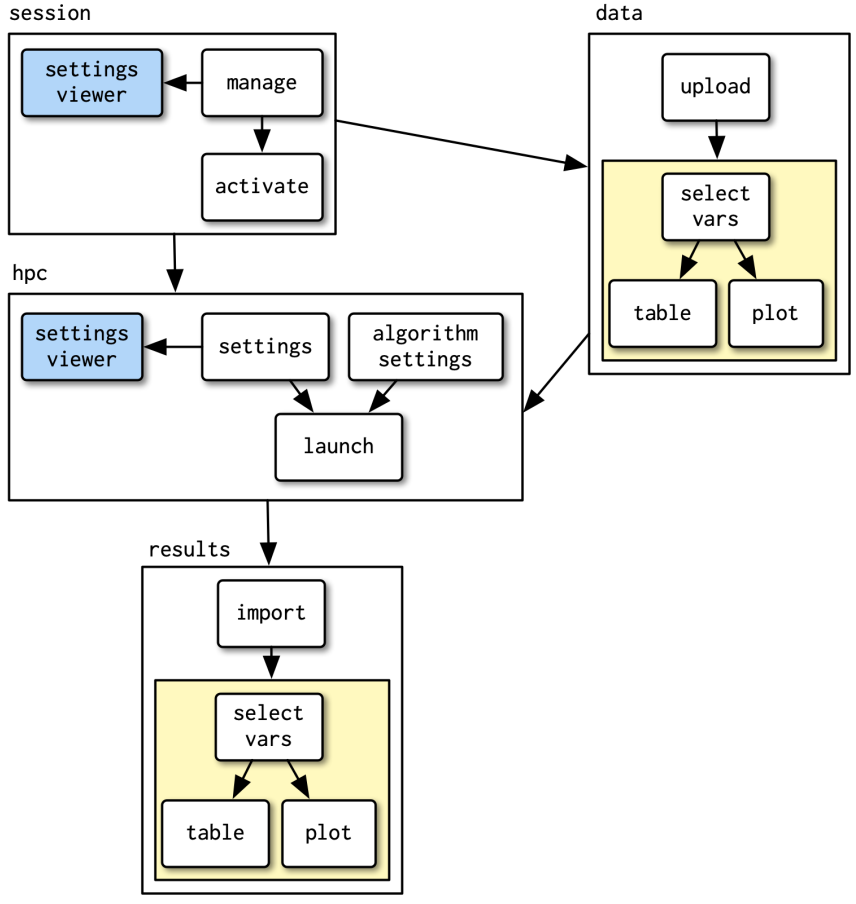
- As Shiny applications grow larger and more complicated, modules are used to manage the growing complexity of Shiny application code.
- Functions are the fundamental unit of abstraction in R, and we designed Shiny to work with them.
- We can write UI-generating functions and call them from our app, and we can write functions to be used in the server function that define outputs and create reactive expressions.

Shiny Modules Workflow

An example of a large and complex Shiny application diagram.



An example of modulerised Shiny application.



Module basics

A module is very similar to an app. Like an app, it's composed of two pieces:

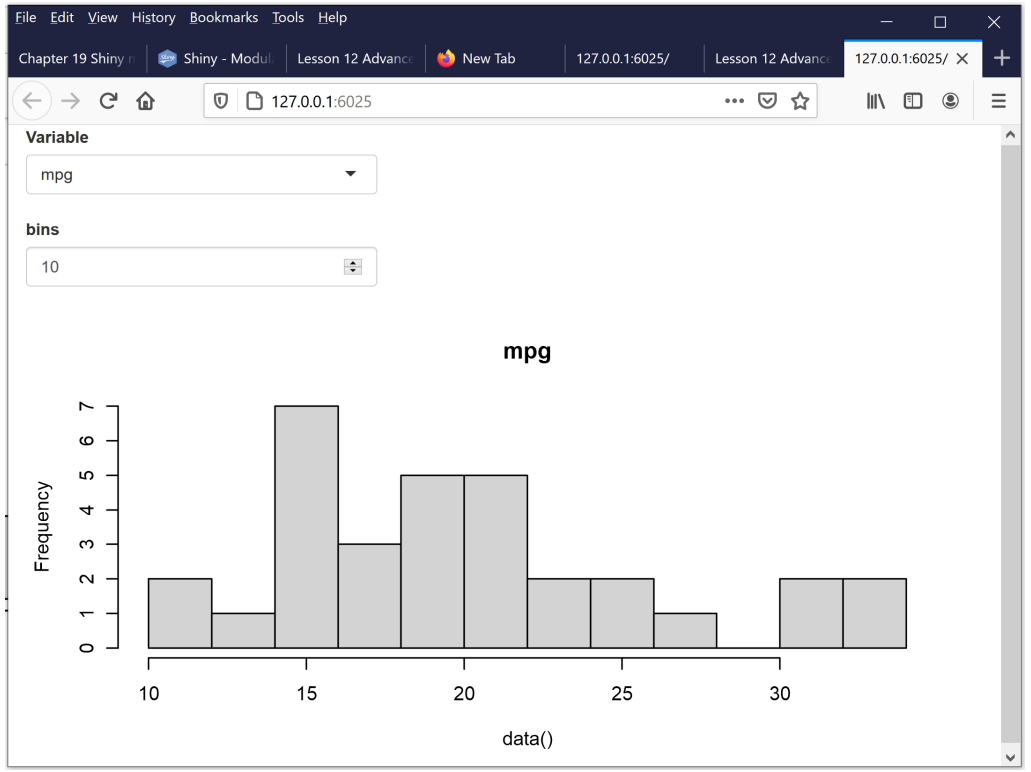
- The **module UI** function that generates the *ui* specification.
- The **module server** function that runs code inside the *server* function.

The two functions have standard forms. They both take an *id* argument and use it to namespace the module. To create a module, we need to extract code out of the app UI and server and put it in to the module UI and server.

In-class Exercise: Building Shiny App with Basic Modele

In order to understand the basics of Shiny modules, let us consider a simple Shiny application codes to plot a histogram shown below.

```
ui <- fluidPage(
  selectInput("var",
              "Variable",
              names(mtcars)),
  numericInput("bins",
               "bins",
               10,
               min = 1),
  plotOutput("hist")
)
server <- function(input,
                   output,
                   session) {
  data <- reactive(mtcars[[input$var]])
  output$hist <- renderPlot({
    hist(data(),
         breaks = input$bins,
         main = input$var)
  }, res = 96)
}
```



Module UI

We'll start with the module UI. There are two steps:

- Put the UI code inside a function that has an id argument.
- Wrap each existing ID in a call to NS(), so that (e.g.) "var" turns into NS(id, "var").

```
histogramUI <- function(id) {
  tagList(
    selectInput(NS(id, "var"), "Variable", choices = names(mtcars)),
    numericInput(NS(id, "bins"), "bins", value = 10, min = 1),
    plotOutput(NS(id, "hist"))
  )
}
```

- Here we have returned the UI components in a *tagList()*, which is a special type of layout function that allows you to bundle together multiple components without actually implying how they will be laid out.
- It is the responsibility of the person calling *histogramUI()* to wrap the result in a layout function like *column()* or *fluidRow()* according to their needs.

Module server

Next we tackle the server function. This gets wrapped inside another function which must have an id argument. This function calls *moduleServer()* with the *id*, and a function that looks like a regular server function:

```
histogramServer <- function(id) {
  moduleServer(id, function(input, output, session) {
    data <- reactive(mtcars[[input$var]])
    output$hist <- renderPlot({
      hist(data(), breaks = input$bins, main = input$var)
    }, res = 96)
  })
}
```

Note that:

- *moduleServer()* takes care of the namespacing automatically: inside of *moduleServer(id)*,
- *input$var* and *input$bins* refer to the inputs with names *NS(id, "var")* and *NS(id, "bins")*.

Revised Shiny Application

Now that we have the ui and server functions, it's good practice to write a function that uses them to generate an app which we can use for experimentation and testing:

```
ui <- fluidPage(
    histogramUI("hist")
    )

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

shinyApp(ui, server)
```

Note that, like all Shiny control, you need to use the same *id* in both UI and server, otherwise the two pieces will not be connected.

In-class Exercise: Function to import csv file

Module UI function

```
# Module UI function
csvFileUI <- function(id, label = "CSV file") {
    # `NS(id)` returns a namespace function, which was save as `ns` and will
    # invoke later.
    ns <- NS(id)

    tagList(
        fileInput(ns("file"), label),
        checkboxInput(ns("heading"), "Has heading"),
        selectInput(ns("quote"), "Quote", c(
            "None" = "",
            "Double quote" = "\"",
            "Single quote" = "'"
        ))
    )
}
```

Module server function

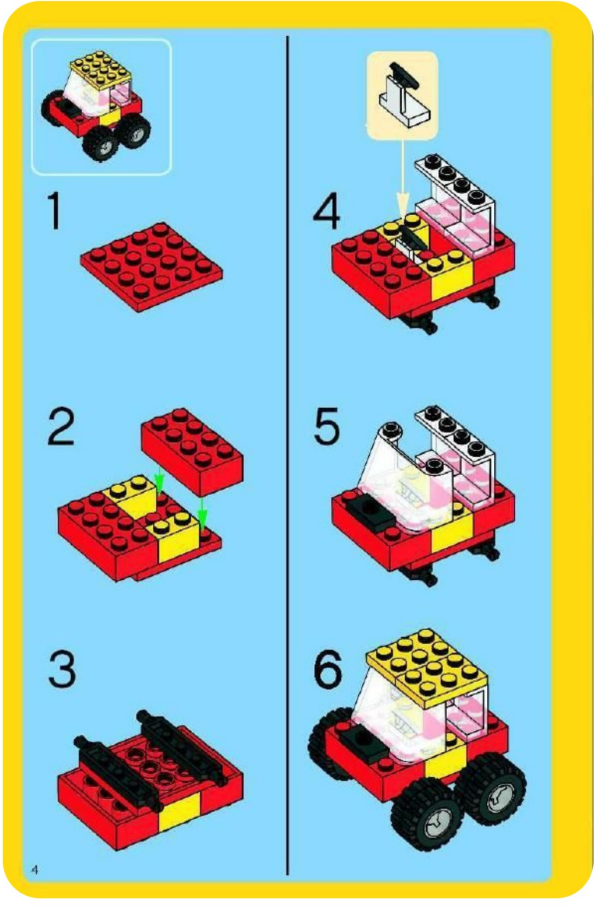
```
csvFileServer <- function(id, stringsAsFactors) {
    moduleServer(
        id,
        function(input, output, session) {
            userFile <- reactive({
                validate(need(input$file, message = FALSE))
                input$file
            })
            dataframe <- reactive({
                read.csv(userFile()$datapath,
                         header = input$heading,
                         quote = input$quote,
                         stringsAsFactors = stringsAsFactors)
            })
            observe({
                msg <- sprintf("File %s was uploaded", userFile()$name)
                cat(msg, "\n")
            })
            return(dataframe)
        }
    )
}
```

The Shiny app

```
ui <- fluidPage(
    sidebarLayout(
        sidebarPanel(
            csvFileUI("datafile", "User data (.csv format)")
        ),
        mainPanel(
            dataTableOutput("table")
        )
    )
)

server <- function(input, output, session) {
    datafile <- csvFileServer("datafile", stringsAsFactors = FALSE)

    output$table <- renderDataTable({
        datafile()
    })
}

shinyApp(ui, server)
```

Programming Shiny Application: Survival Tip!

What can we learn from Lego?



- Sketch the storyboard
- Building the app incrementally
  - Using prototyping approach
  - Start as simple as possible
  - Adding features one at a time
- Save -> Run App

Debugging

Programming == Frustration?

Source: The original article.

Three main cases of problems which we'll discuss below:

- You get an unexpected error. This is the easiest case, because you'll get a traceback which allows you to figure out exactly where the error occurred.
- You don't get any errors, but some value is incorrect. Here, you'll need to use the interactive debugger, along with your investigative skills to track down the root cause.
- All the values are correct, but they're not updated when you expect. This is the most challenging problem because it's unique to Shiny, so you can't take advantage of your existing R debugging skills.

Common errors

"Object of type 'closure' is not subsettable"

- You forgot to use () when retrieving a value from a reactive expression *plot(userData)* should be *plot(userData())*

Common errors

"Unexpected symbol"

"Argument xxx is missing, with no default"

- Missing or extra comma in UI.
- Sometimes Shiny will realise this and give you a hint, or use RStudio editor margin diagnostics.

Common errors

"Operation not allowed without an active reactive context. (You tried to do something that can only be done from inside a reactive expression or observer.)

- Tried to access an input or reactive expression from directly inside the server function. You must use a reactive expression or observer instead.
- Or if you really only care about the value of that input at the time that the session starts, then use isolate().

Standard R debugging tools

- Tracing
  - tracebacks
  - print()/cat()/str()
  - renderPrint eats messages, must use cat(file = stderr(), ""...)
  - Also consider shinyjs package's logjs, which puts messages in the browser's JavaScript console
- Debugger
  - Set breakpoints in RStudio
  - browser()
  - Conditionals: if (!is.null(input$x)) browser()

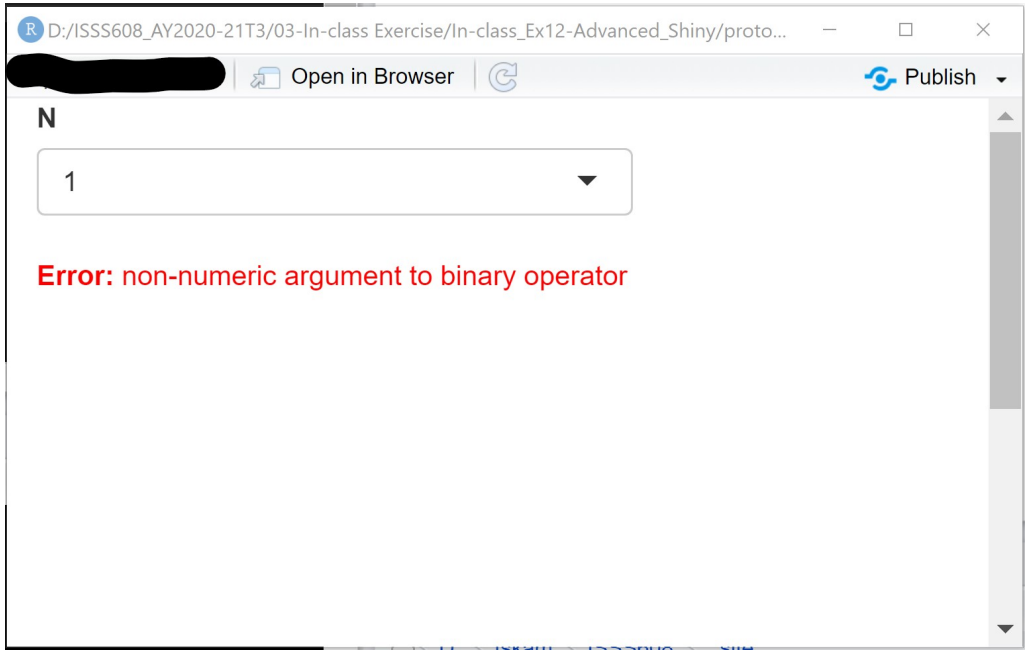
In-class Exercise: Working with Tracebacks in Shiny

In this hands-on exercise, you will learn how to detect error by using tracebacks in Shiny.

The codes:

```
library(shiny)

f <- function(x) g(x)
g <- function(x) h(x)
h <- function(x) x * 2

ui <- fluidPage(
    selectInput("n", "N", 1:10),
    plotOutput("plot")
)
server <- function(input, output, session) {
    output$plot <- renderPlot({
        n <- f(input$n)
        plot(head(cars, n))
    }, res = 96)
}
shinyApp(ui, server)
```

Op, there is an error!



Learning how to debug

Tracebacks in Shiny

```
Listening on http://127.0.0.1:7775
Warning: Error in *: non-numeric argument to binary operator
  169: g [D:\ISSS608_AY2020-21T3\03-In-class Exercise\In-class_E
x12-Advanced_Shiny\prototype7a/app.R#4]
  168: f [D:\ISSS608_AY2020-21T3\03-In-class Exercise\In-class_E
x12-Advanced_Shiny\prototype7a/app.R#3]
  167: renderPlot [D:\ISSS608_AY2020-21T3\03-In-class Exercise\I
n-class_Ex12-Advanced_Shiny\prototype7a/app.R#13]
  165: func
  125: drawPlot
  111: <reactive:plotObj>
   95: drawReactive
   82: renderFunc
   81: output$plot
    1: runApp
```

Things to learn from the call stack:

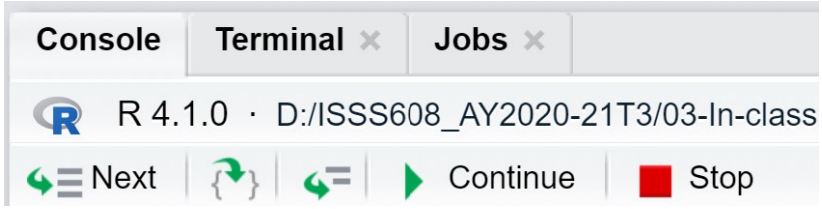
- The first few calls start the app in this case you just see runApp() but depending on how you start the app, you might see something more complicated.
- In general, you can ignore anything before the first runApp(); this is just the setup code to get the app running.
- Next, you'll see some internal Shiny code in charge of calling the reactive expression:
  - Here, spotting **output$plot** is really important — that tells which of your **reactives (plot)** is causing the error. The next few functions are internal, and you can ignore them.
- Finally, at the very top, you'll see the code that you have written:
  - This is the code called inside of *renderPlot()*. You can tell you should pay attention here because of the file path and line number; this lets you know that it's your code.

If you get an error in your app but don't see a traceback then make sure that you're running the app using Cmd/Ctrl + Shift + Enter (or if not in RStudio, calling runApp()), and that you've saved the file that you're running it from. Other ways of running the app don't always capture the information necessary to make a traceback.

In-class Exercise: Working with RStudio's Interactive Debugger

In this hands-on Exercise, you will learn how to work with the interactive debugger in RStudio.

- Double click prototype7a.
- Double click on *prototype7a.rproj* file to open the project file in RStudio.
- Click on *app.R* file to open the Shiny app file on RStudio
- Add a call to *browser()* in your source code (for example line 5).
- Click in **Run App** button to run the Shiny app.



- Next (press n): executes the next step in the function. Note that if you have a variable named n, you'll need to use print(n) to display its value.
- Continue (press c): leaves interactive debugging and continues regular execution of the function. This is useful if you've fixed the bad state and want to check that the function proceeds correctly.
- Stop (press Q): stops debugging, terminates the function, and returns to the global workspace. Use this once you've figured out where the problem is, and you're ready to fix it and reload the code.

shinytest

- Shinytest uses snapshot-based testing strategy.
- The first time it runs a set of tests for an application, it performs some scripted interactions with the app and takes one or more snapshots of the application's state.
- These snapshots are saved to disk so that future runs of the tests can compare their results to them.

Testing

- There are many possible reasons for an application to stop working. These reasons include:
  - An upgraded R package has different behavior. (This could include Shiny itself!)
  - You make modifications to your application.
  - An external data source stops working, or returns data in a changed format.
- Automated tests can alert you to these kinds of problems quickly and with almost zero effort, after the tests have been created.

References

Shiny Layout

- Application layout guide

Shiny Module

- Chapter 19 Shiny modules of Mastering Shiny.
- Modularizing Shiny app code, online article
- Communication between modules. This is a relatively old article, some functions have changed.
- Shiny Modules
- Shiny Modules (part 1) : Why using modules?
- Shiny Modules (part 2): Share reactive among multiple modules
- Shiny Modules (part 3): Dynamic module call