Interactive Applications with `Shiny`

ZevRoss
Know Your Data

Shiny: A web application framework for R

Why Shiny?

- Interactively view data
- Write your app in R
- Relatively quick to create simple apps

Shiny resources

- RStudio's shiny site
- Dean Attali's interactive tutorial
- My blog post with 40 example apps

Creating a shiny application

A Shiny application requires two things

- User interface (the beauty)
- Server (the brains)

A very simple Shiny app (code)

```r
library(shiny)
```

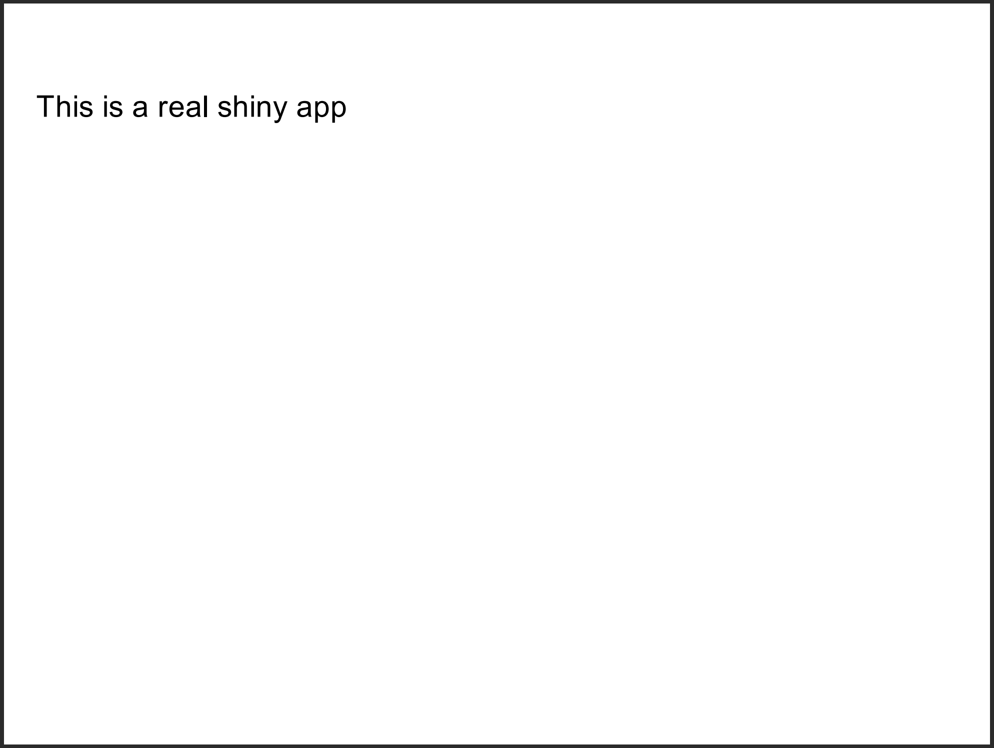
```r
ui <- basicPage("This is a real shiny app")
```

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

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

A very simple Shiny app (app)

```r
library(shiny)
ui <- basicPage("This is a real shiny app") # UI
server <- function(input, output, session) { } # server
shinyApp(ui = ui, server = server)
```

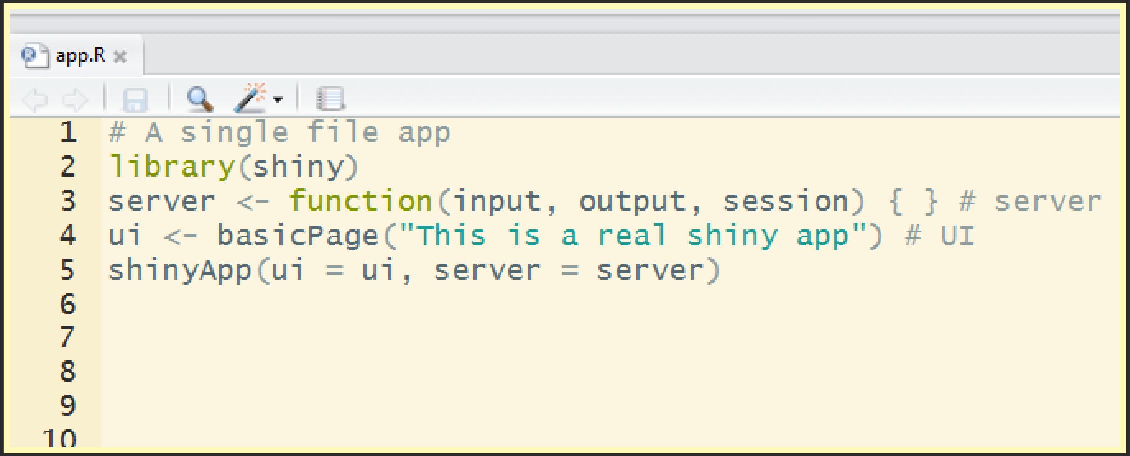


This is a real shiny app

Single-file vs multi-file Shiny apps

Single-file app

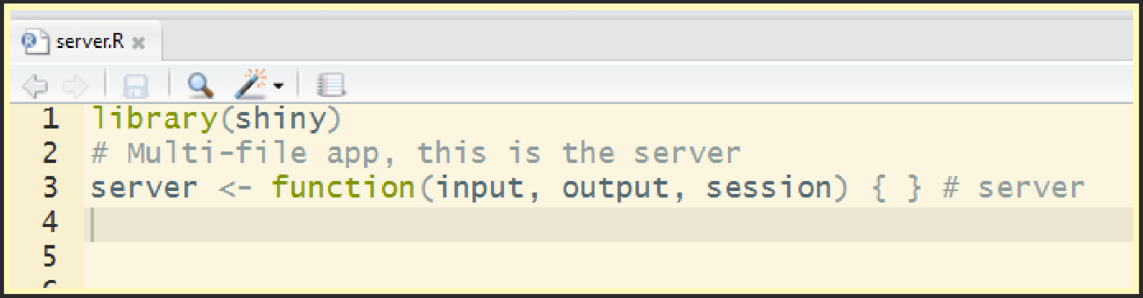
For smaller apps. Use the `shinyApp` function to launch:



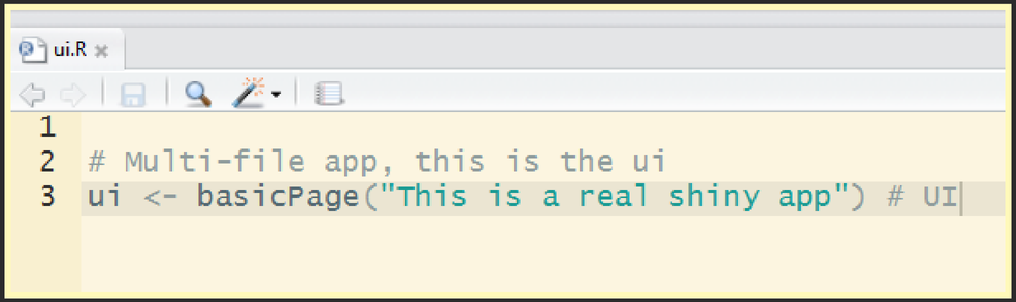
```r
# A single file app
library(shiny)
server <- function(input, output, session) { } # server
ui <- basicPage("This is a real shiny app") # UI
shinyApp(ui = ui, server = server)
```

Multi-file app

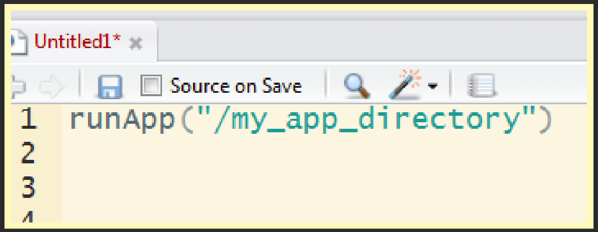
For larger apps. Use the runApp function to launch:



```r
library(shiny)
# Multi-file app, this is the server
server <- function(input, output, session) { } # server
```



```r

# Multi-file app, this is the ui
ui <- basicPage("This is a real shiny app") # UI
```



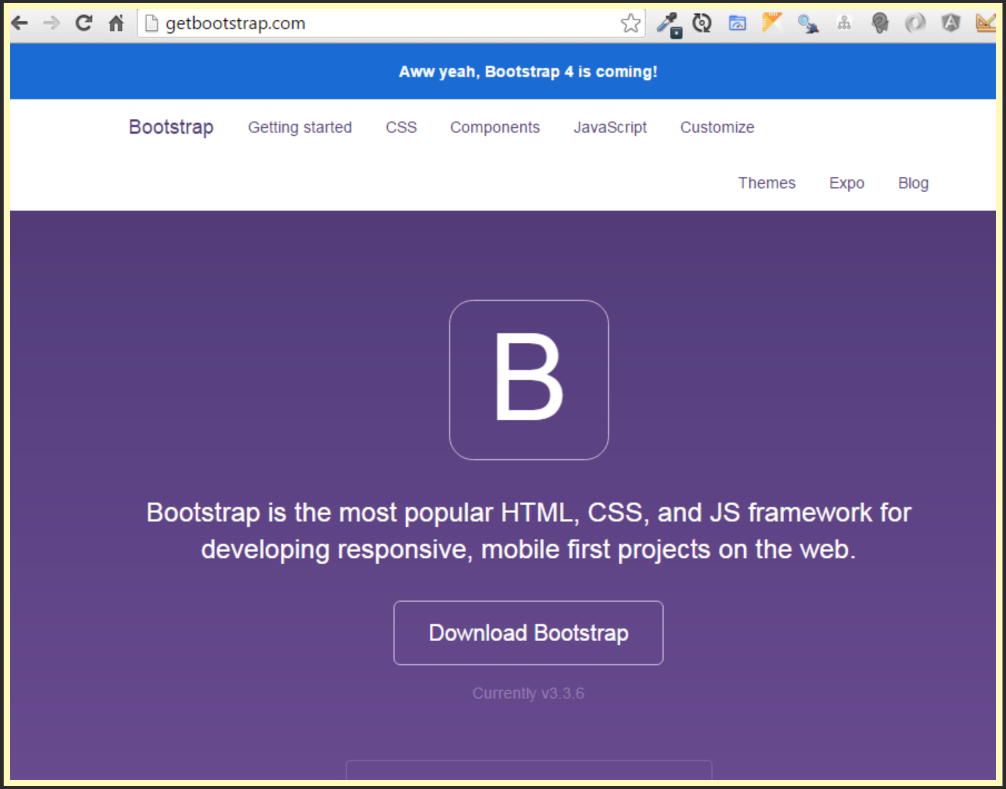
```r
runApp("/my_app_directory")
```

User Interface (UI)

UI key topics

- Layout
- Adding HTML objects like titles and paragraphs
- Widgets (sliders, text boxes etc)
- Styles

Shiny's defaults are based on Twitter Bootstrap



Bootstrap is the most popular HTML, CSS, and JS framework for developing responsive, mobile first projects on the web.

UI Layout

Helpful layout reference

- RStudio has a nice page describing layout options

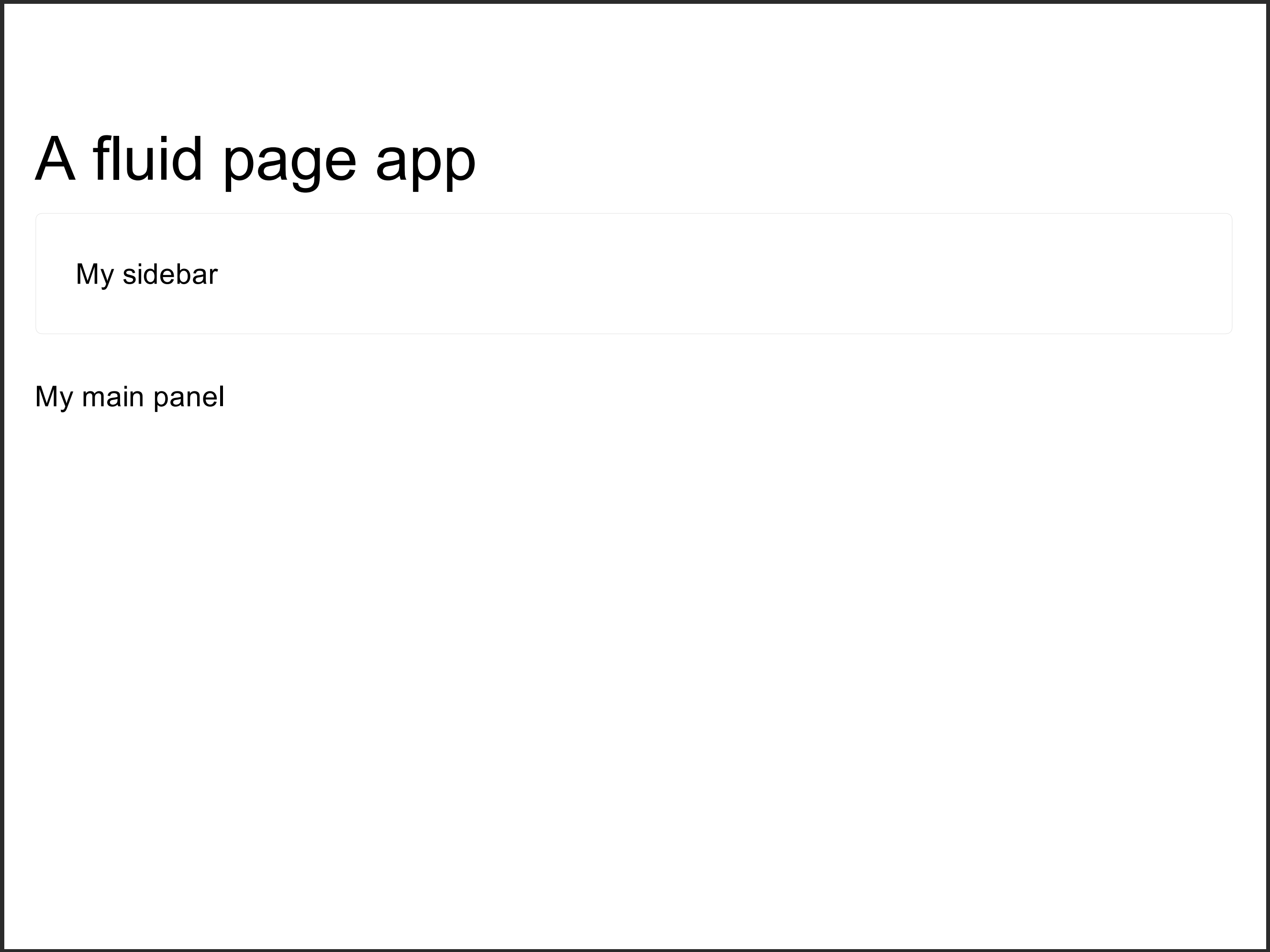
Shiny has pre-defined layout functions

- `basicPage`
- `fluidPage`
- `sidebarLayout`
- `navbarPage`

An example of pre-defined layout functions (code)

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

  titlePanel("A fluid page app"),

  sidebarLayout(

    sidebarPanel(
      "My sidebar"
    ),

    mainPanel(
      "My main panel"
    )
  )
)
server <- function(input, output, session) { }
shinyApp(ui = ui, server = server)
```

An example of pre-defined layout functions (app)



A fluid page app

My sidebar

My main panel

Note that as a fluid layout the "sidebar" may span the entire page on smaller screens (like in the PDF of these slides)

Add HTML tags

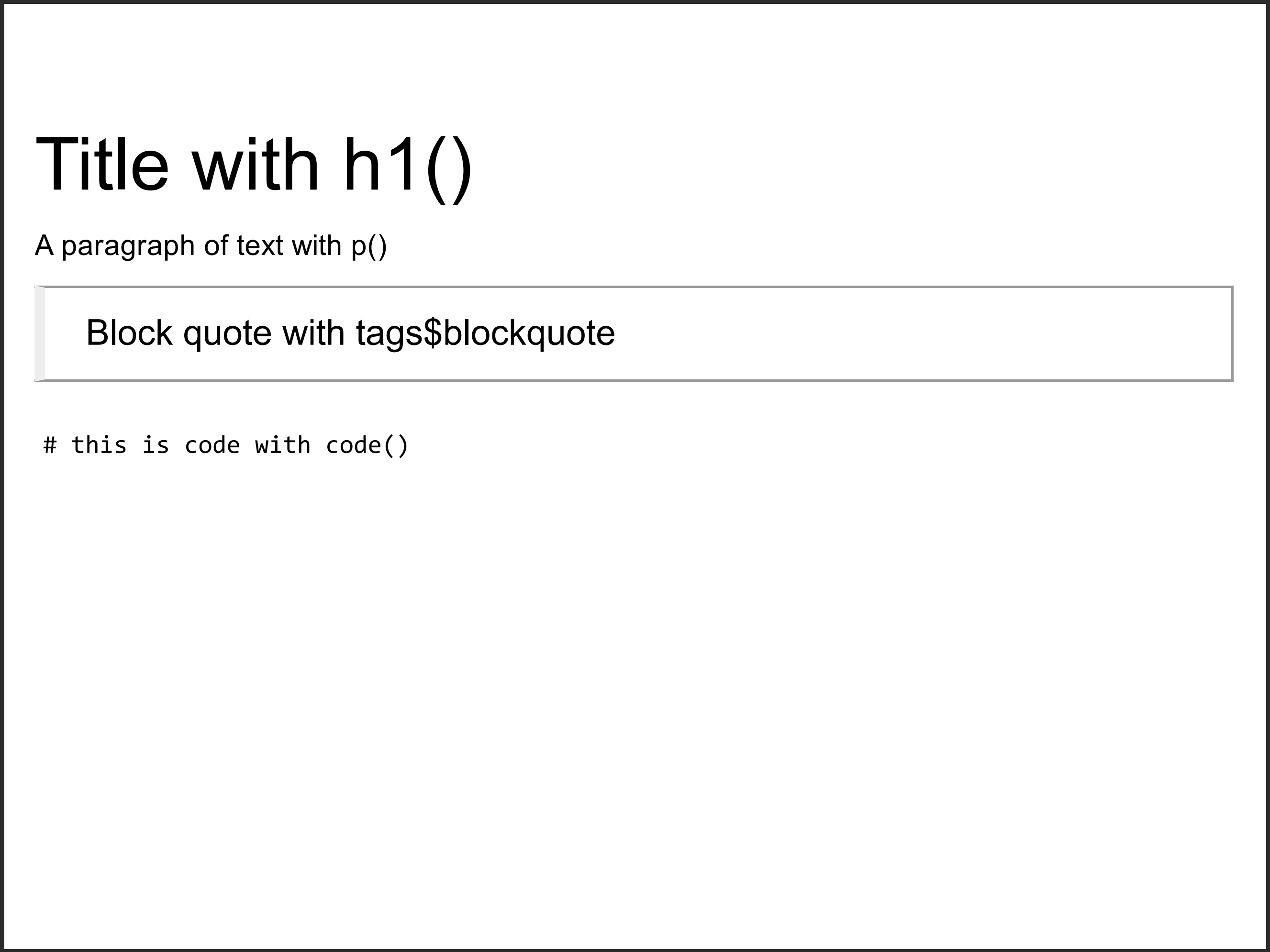
- HTML tags can be included with `tags$` (e.g., `tags$h1`, `tags$blockquote`)

- For common HTML tags you can use the tag name as a function (e.g., `h1()`, `p()`)

Adding HTML tags (code)

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

  h1("Title with h1()"),
  p("A paragraph of text with p()"),
  tags$blockquote("Block quote with tags$blockquote"),
  code('# this is code with code()')

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

Adding HTML tags (app)



Title with h1()

A paragraph of text with p()

> Block quote with tags$blockquote

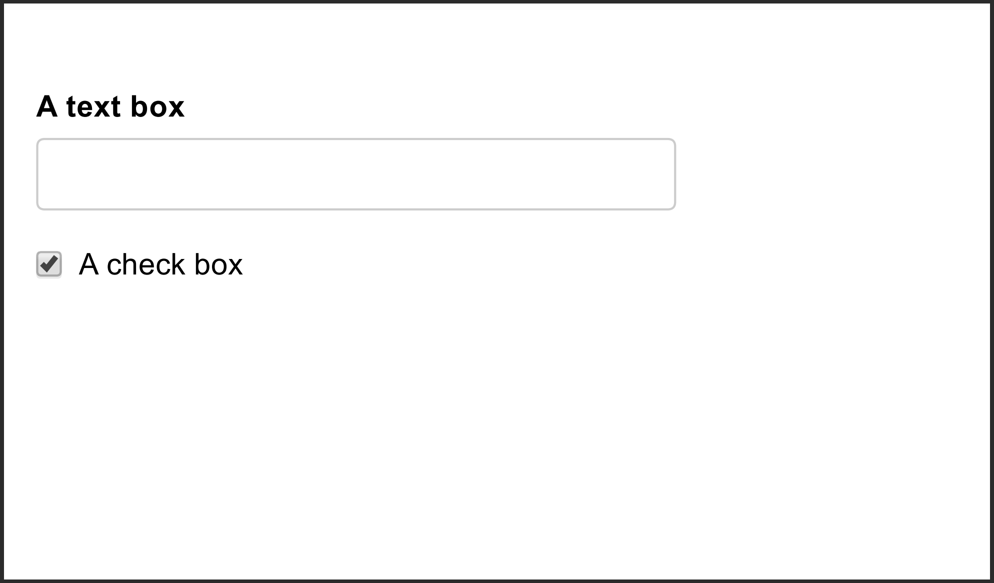
```
# this is code with code()
```

Widgets for user interaction

- Functions for adding widgets, `sliderInput`, `textInput` etc
- RStudio has a **widget gallery** with examples

Adding a widget to the UI is easy

```r
library(shiny)
ui <- fluidPage(
  textInput(inputId = "txt", "A text box"),
  checkboxInput(inputId = "chk", "A check box", TRUE)
)
server <- function(input, output, session) { }
shinyApp(ui = ui, server = server)
```



Adding style

Styles are added with the style language of the web — CSS.

Best practice is to keep all styles in a single style sheet

Read a style sheet with `includeCSS`

```r
ui <- fluidPage(

  includeCSS("path-to-style/style.css")

)
```

You can also manually define styles

Header and inline styles (code)

```r
ui <- basicPage(
  # styles in the header
  tags$head(
    tags$style(HTML("
      body {
        background-color: cornflowerblue;
        color: Maroon;
      }
    "))
  ),
  # here is an in-line style
  h3(style="color:white", "CSS using the HTML tag"),
  p("Some important text")

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

Header and inline styles (app)

CSS using the HTML tag

Some important text

A note for slides: print to PDF is not rendering the shiny styles so this app looks white. Try the code for yourself to see the color

A final UI note about commas

In the UI you need to separate multiple items at the same level with commas. The server is a traditional R function so no commas are necessary to separate lines.

```r
ui <- basicPage(
  h1("A title"),
  h4("A subtitle"),
  p("A paragraph")
)
```

```r
ui <- basicPage(
  tabsetPanel(
    tabPanel("a",
             h1("title"),
             h4("title2")
    ),
    tabPanel("b"),
    tabPanel("c")
  )
)
```

exercise 2 (just questions 1-7, the shiny ui)

Shiny Server

Server key topics

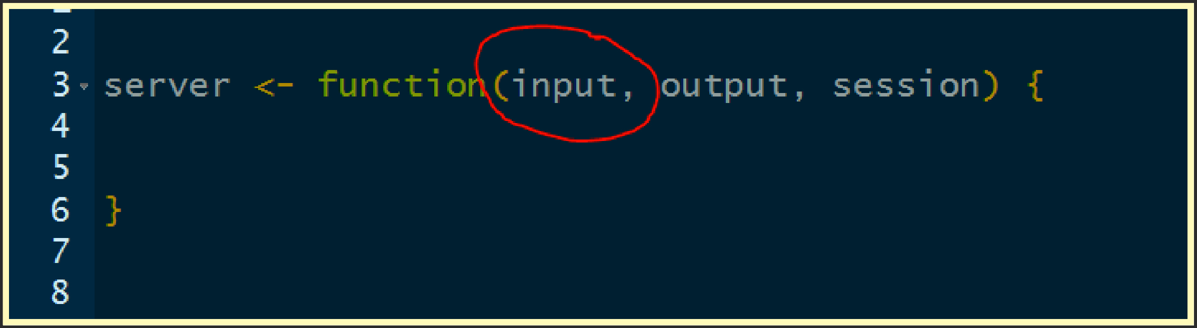
- Reactive values
- The listeners `observe` and `reactive`
- Generating output (text, tables, plots)

Reactive values

What is the ID of this text box?

```
ui <- fluidPage(

  textInput("mytextbox", "A text box")

)
```

All of the widget IDs get added to the input list used in the server



```

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


}


```

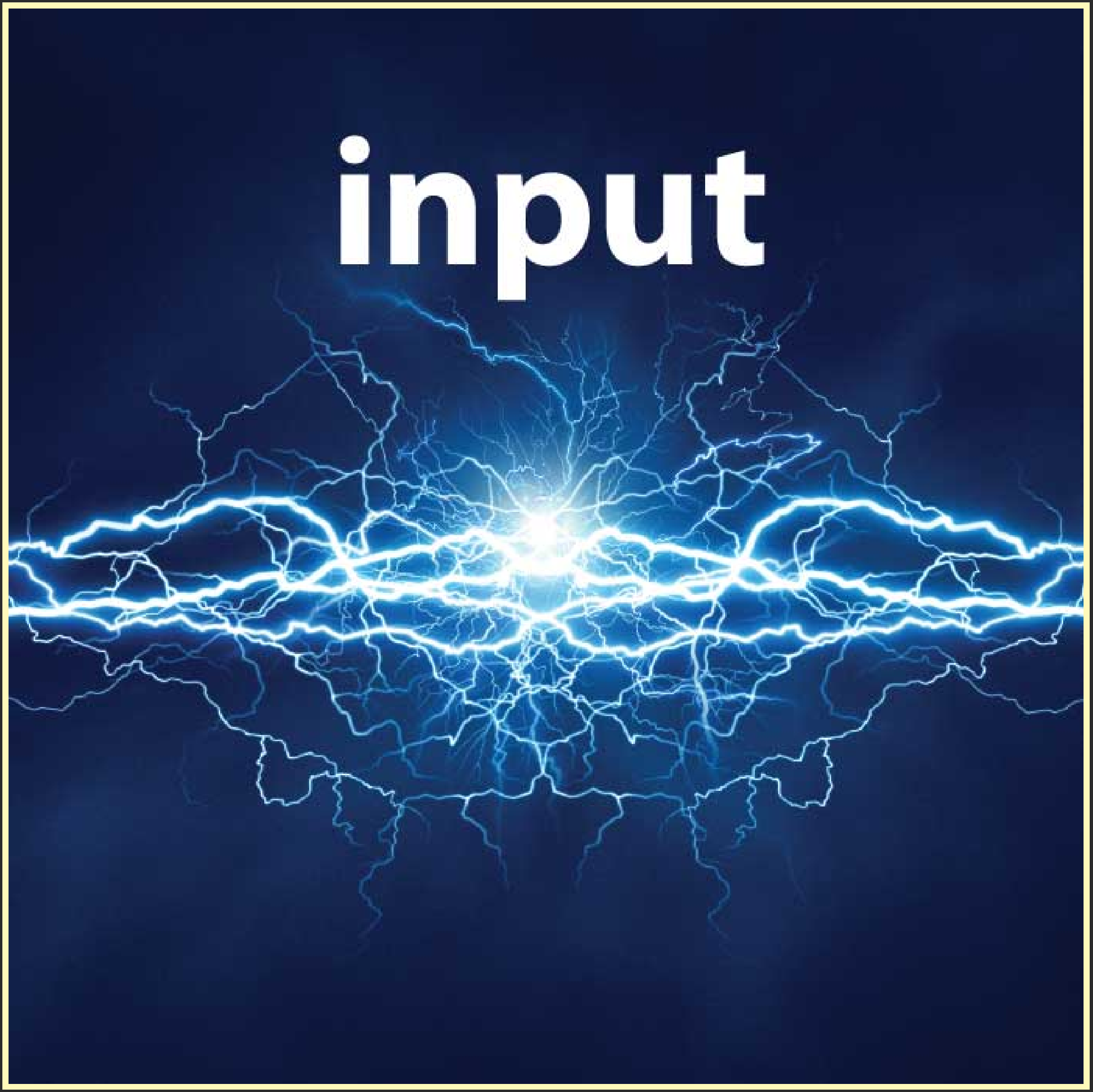
Our server can then find the text box values with

```
input$mytextbox
```

So you might think that this would work

```r
ui <- basicPage(
  textInput("mytextbox", "A text box")
)

server <- function(input, output, session) {
  print(input$mytextbox)
}
```

But input is a special kind of list



Input can only be read by a "reactive expression"

- `input` is a list of reactive values
- Reactive values can only be handled by functions designed to handle them
- These functions, or "reactive expressions", include `observe`, `reactive` and the `render*` functions

The listeners `observe` **and** `reactive`

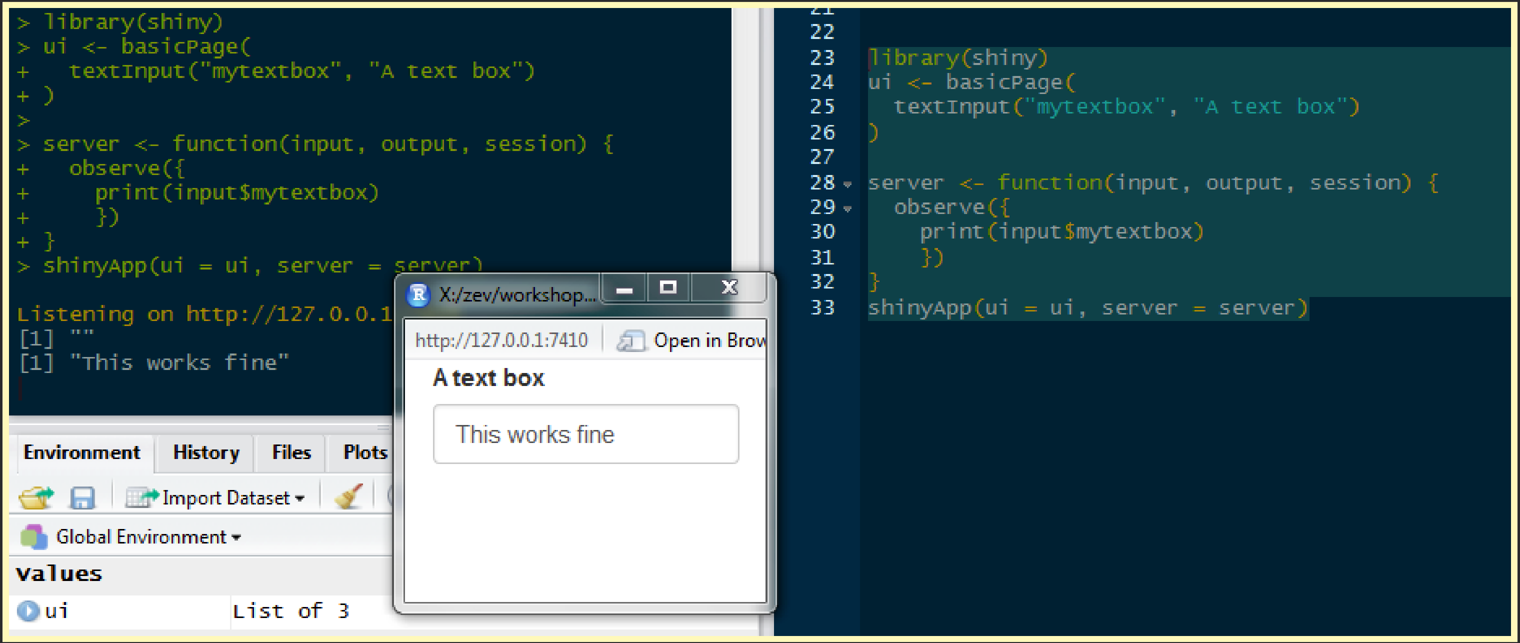
Introducing `observe`

A function to generate side effects (but does not return a value) based on user input.

Revisit the `print` example

```r
ui <- basicPage(
  textInput("mytextbox", "A text box")
)

server <- function(input, output, session) {
  observe({
    print(input$mytextbox)
    })
}
```

The `print` example in action



```
> library(shiny)
> ui <- basicPage(
+   textInput("mytextbox", "A text box")
+ )
>
> server <- function(input, output, session) {
+   observe({
+     print(input$mytextbox)
+     })
+ }
> shinyApp(ui = ui, server = server)

Listening on http://127.0.0.1
[1] ""
[1] "This works fine"
```

```
library(shiny)
ui <- basicPage(
  textInput("mytextbox", "A text box")
)

server <- function(input, output, session) {
  observe({
    print(input$mytextbox)
    })
}
shinyApp(ui = ui, server = server)
```

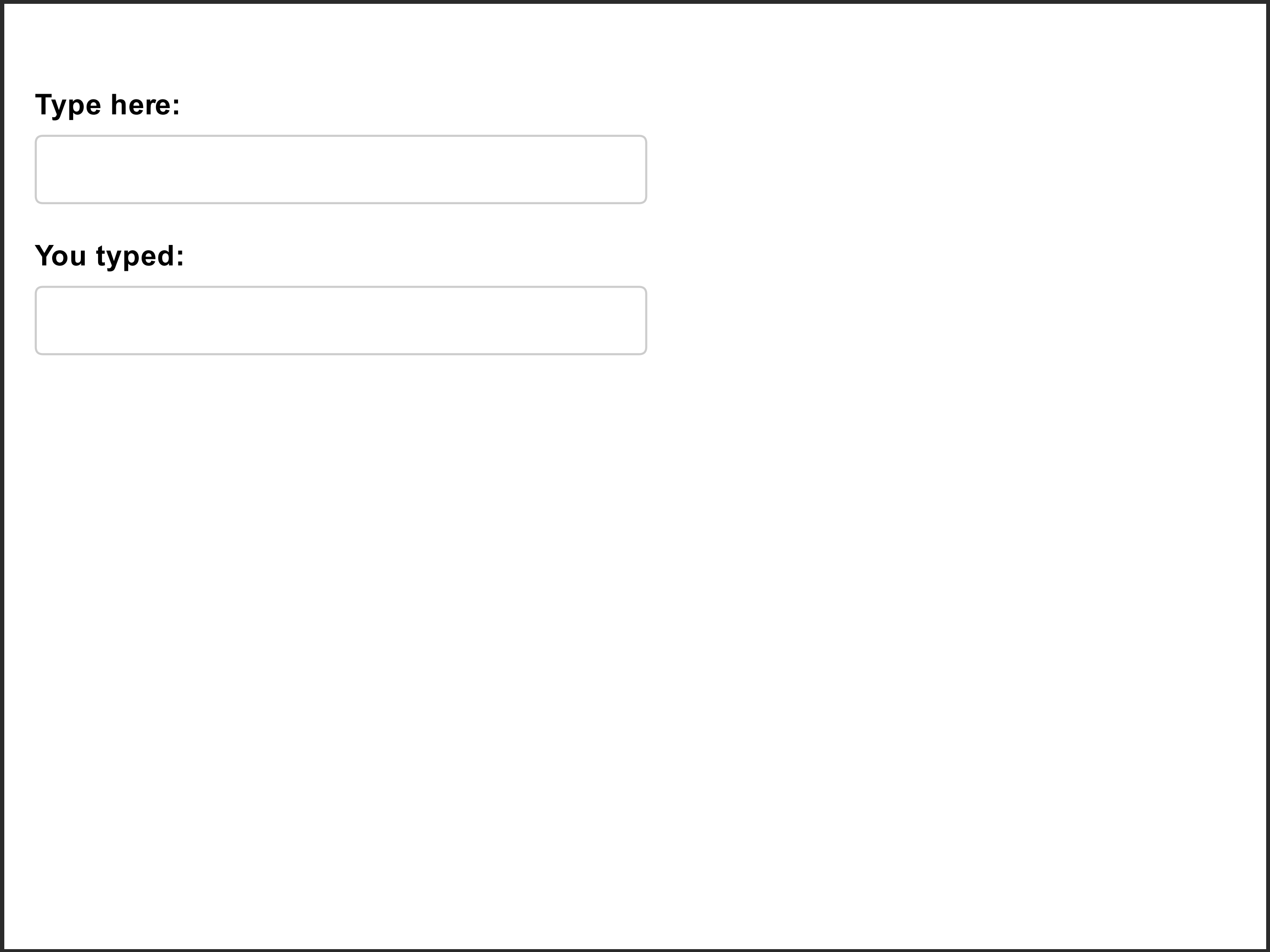
A change to the text box triggers the code in `observe` to run

- Joe Cheng calls `observe` "eager". When its dependencies change, it executes right away.

Eager `observe` example (code)

```r
ui <- fluidPage(
  textInput(inputId = "txt1", "Type here:"),
  textInput(inputId = "txt2", "You typed:")
)

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

    updateTextInput(session, "txt2", value = input$txt1)

  })
}

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

Eager `observe` example (app)


Type here:

You typed:

`observe` does not return values

- The `observe` function is designed cause side effects (on purpose) but not return a value.
- What if we want to do calculations and return a value based on user input?

Introducing `reactive`

- Operates a lot like a function
- Can be called and returns a value
- Lazy, not eager, doesn't execute until called

An easy way to remember

Keep your side effects
Outside of your reactives
Or I will kill you

—Joe Cheng

A `reactive` to generate output

Set up a `reactive` called my_results:

```r
server <- function(input, output, session) {
  my_results <- reactive({
    iris[input$myrow, "Species"]
  })
}
```

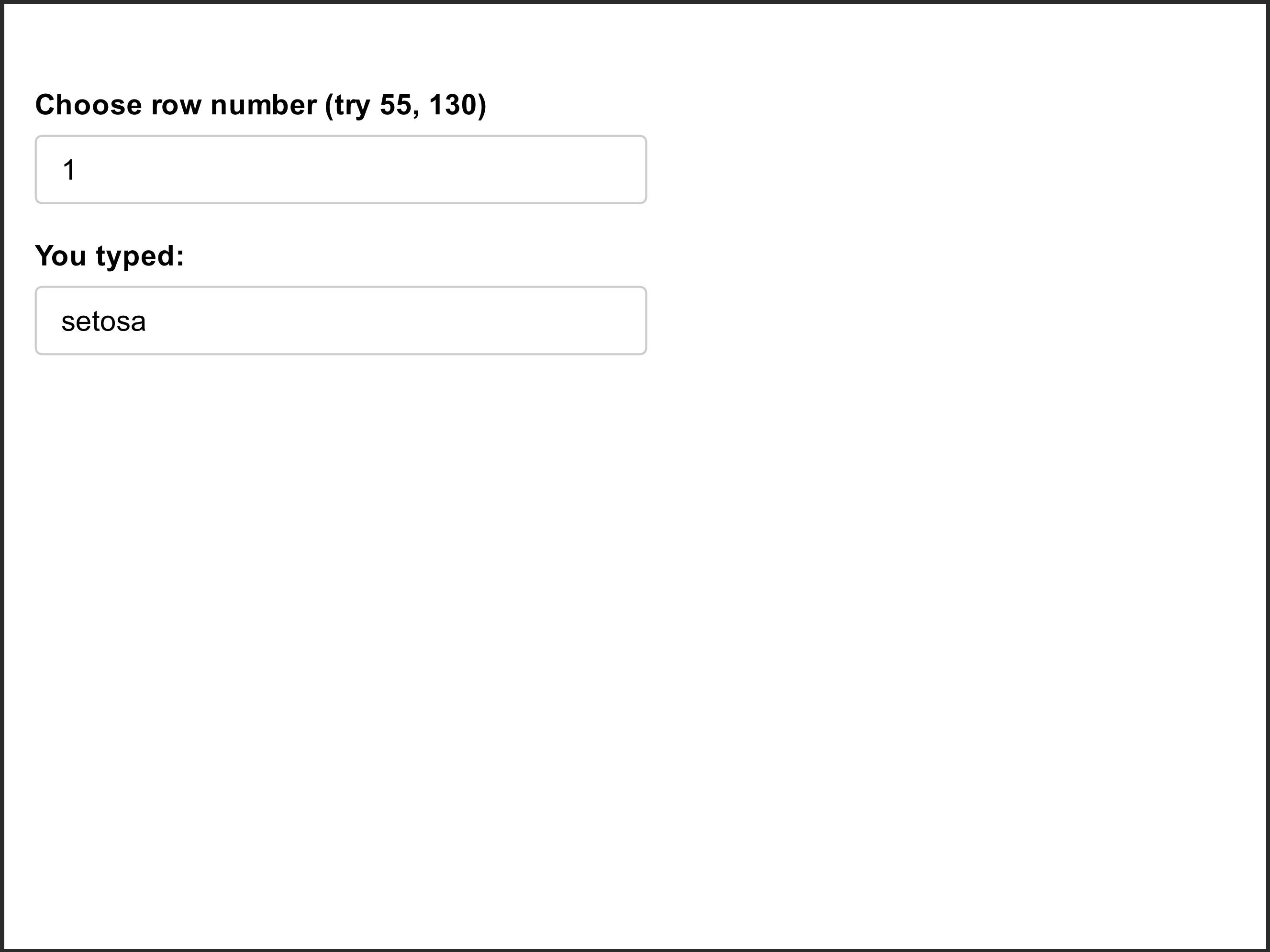
Run the `reactive` in your server like a function (but must be within an `observe` or `render*` function):

```r
my_results()
```

reactive in action (code)

```
ui <- fluidPage(
  numericInput("myrow", "Choose row number (try 55, 130)", 
1),
  textInput(inputId = "txt2", "You typed:")
)

server <- function(input, output, session) {
  my_results <- reactive({

    iris[input$myrow, "Species"]

  })

  observe({
    input$myrow
    updateTextInput(session, "txt2", value = my_results())
  })

}

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

`reactive` in action (app)



Choose row number (try 55, 130)

1

You typed:

setosa

One more note about `observe` and `reactive`

Which checkbox triggers the `observe` code to run?

```r
ui <- basicPage(

  checkboxInput("chk1", "Check 1",  FALSE),
  checkboxInput("chk2", "Check 2",  FALSE)

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

  # changes to either chk1 or chk2 trigger the code to run
  observe({
    print(input$chk1)
    print(input$chk2)
  })

}
```

Prevent unwanted reactions with `observeEvent` and `eventReactive`

```r
ui <- basicPage(

  checkboxInput("chk1", "Check 1",  FALSE),
  checkboxInput("chk2", "Check 2",  FALSE)

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

  # only changes to chk1 trigger the code to run
  observeEvent(input$chk1, {
    print(input$chk1)
    print(input$chk2)
  })

}
```

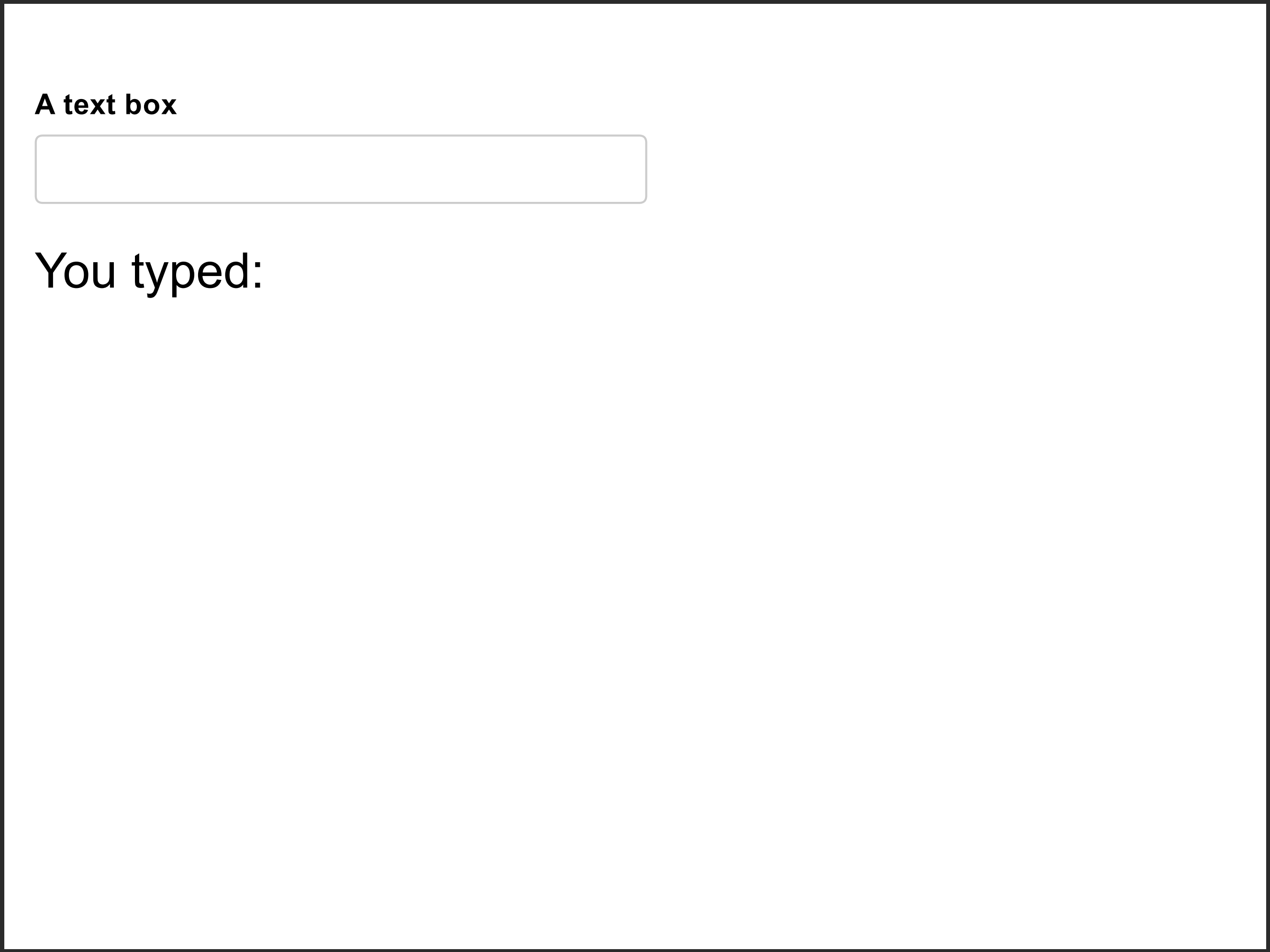
exercise 2 (just questions 8-11, the shiny ui)

Generating dynamic output (text, plots, tables)

Setup for dynamic output (code)

```r
ui <- basicPage(
  textInput(inputId = "txt", "A text box"),
  h3(style="color:green", "You typed:")
)
server <- function(input, output, session) { }
shinyApp(ui = ui, server = server)
```

Setup for dynamic output (app)



A text box

You typed:

A note for slides: print to PDF is not rendering the shiny styles so the green text looks black. Try the code for yourself to see the color

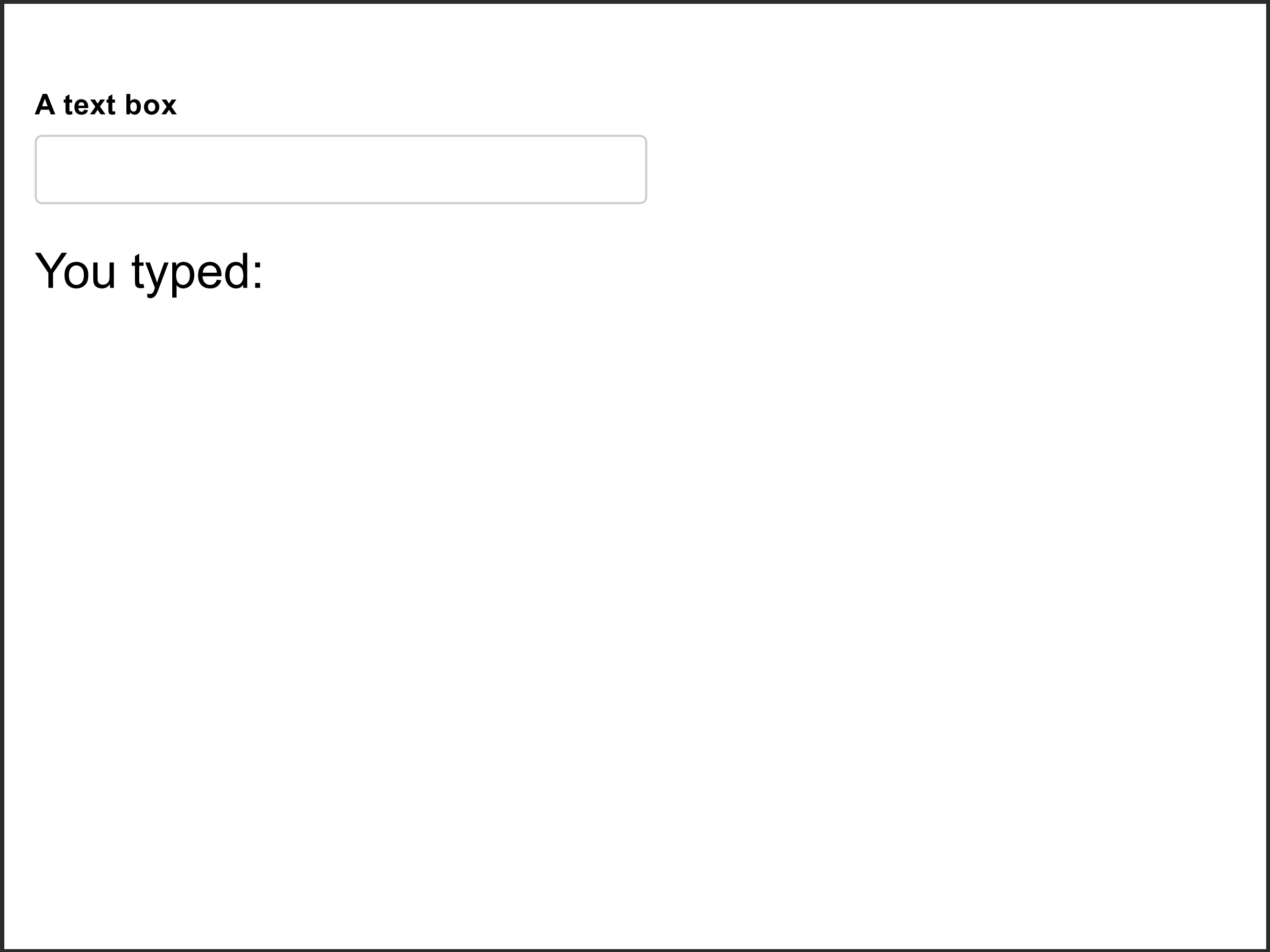
To generate dynamic text, tables and plots you need two pieces

- You need a `render*` function in the server (e.g., `renderText`)
- You need a `*Output` function in the UI (e.g., `textOutput`)

renderText to textOutput (code)

```
ui <- basicPage(
  textInput(inputId = "txt", "A text box"),
  h3(style="color:green", "You typed:"),
  textOutput("usertext") # here is our UI output
)
server <- function(input, output, session) {

  output$usertext <- renderText({
    input$txt # return text box value
  }) # our render function

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

`renderText` **to** `textOutput` **(app)**



A text box

You typed:

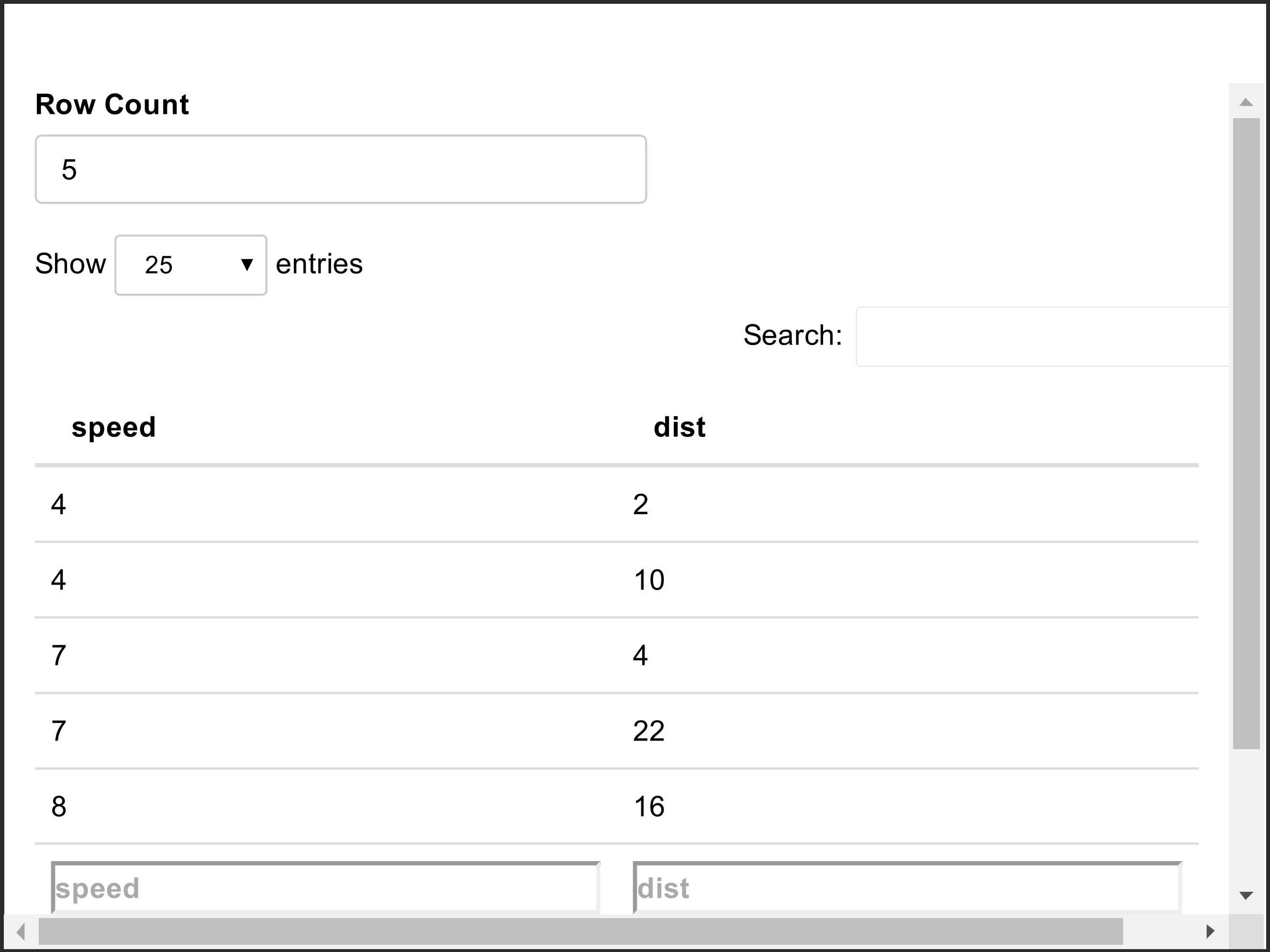
How about a dynamic table?

- Use `renderDataTable` in the server
- Use `dataTableOutput` in the UI

A dynamic table (code)

```r
ui <- basicPage(
  numericInput(inputId = "num", "Row Count", value=5),
  dataTableOutput("newtable") # output to user
)
server <- function(input, output, session) {

  output$newtable <- renderDataTable({
    cars[1:input$num,]
  }) # render a data table

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

A dynamic table (app)



Row Count

5

Show 25 entries

Search:

speed	dist
4	2
4	10
7	4
7	22
8	16
speed	dist

Remember `ggplot2`?

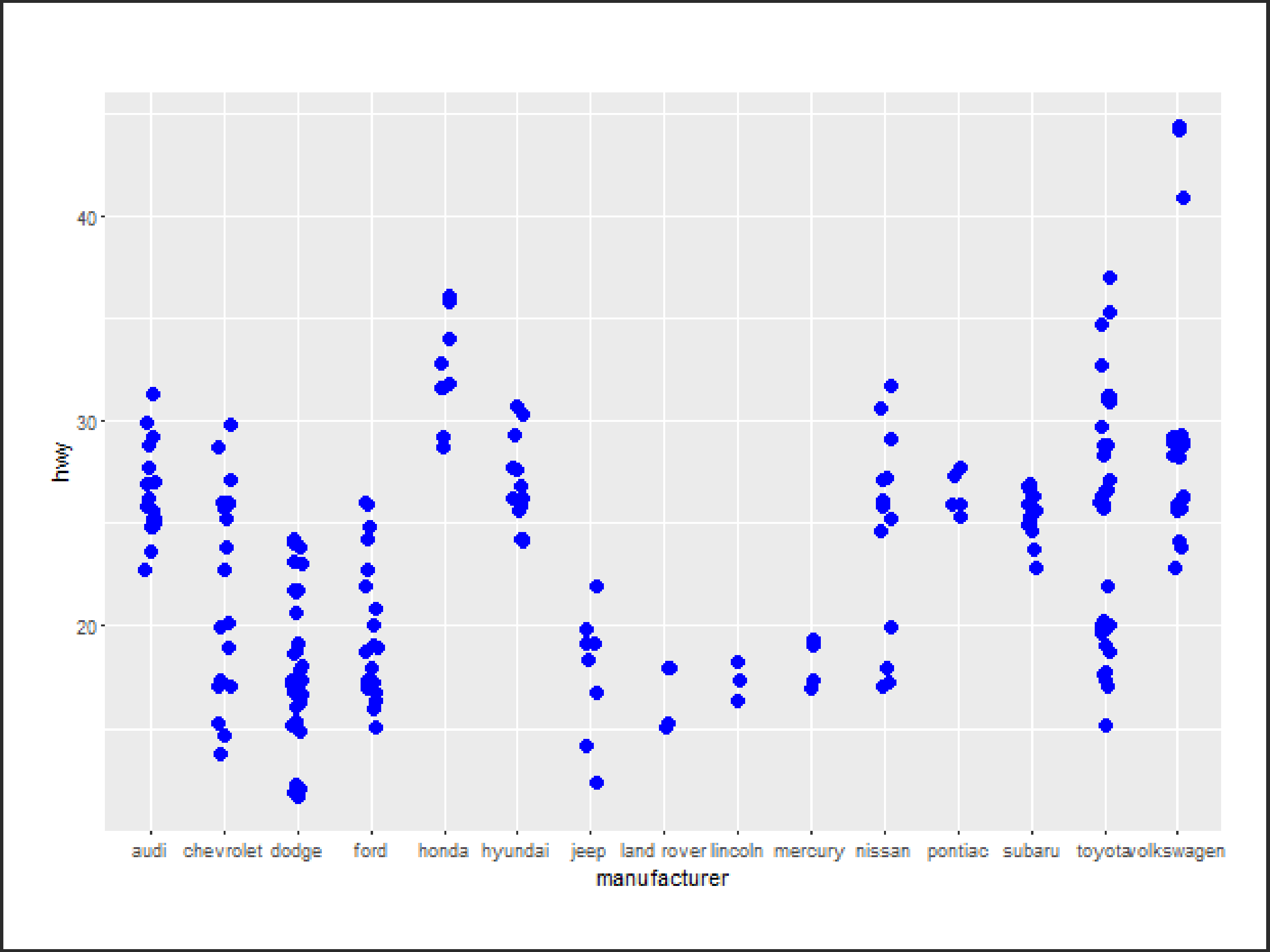
Include a plot in an app

- `renderPlot` in the server
- `plotOutput` in the UI

A ggplot in shiny (code)

```r
library(ggplot2)
ui <- basicPage(
  plotOutput("myplot") # output plot to user
)

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

  output$myplot <- renderPlot({

    ggplot(mpg, aes(manufacturer, hwy)) +
      geom_jitter(color="blue", width=0.2, size=3)

  }) # render a plot for the UI
}
shinyApp(ui = ui, server = server)
```

A ggplot in shiny (app)



Make the ggplot a little more fun

Allow the user to select the car manufacturer (the car "make")

Add a manufacturer selector (the UI)

```r
# Add this to the UI
selectInput("make", "Choose make",
            multiple = TRUE, choices=mpg$manufacturer,
            selected="toyota")
```

Add a manufacturer selector (the server)

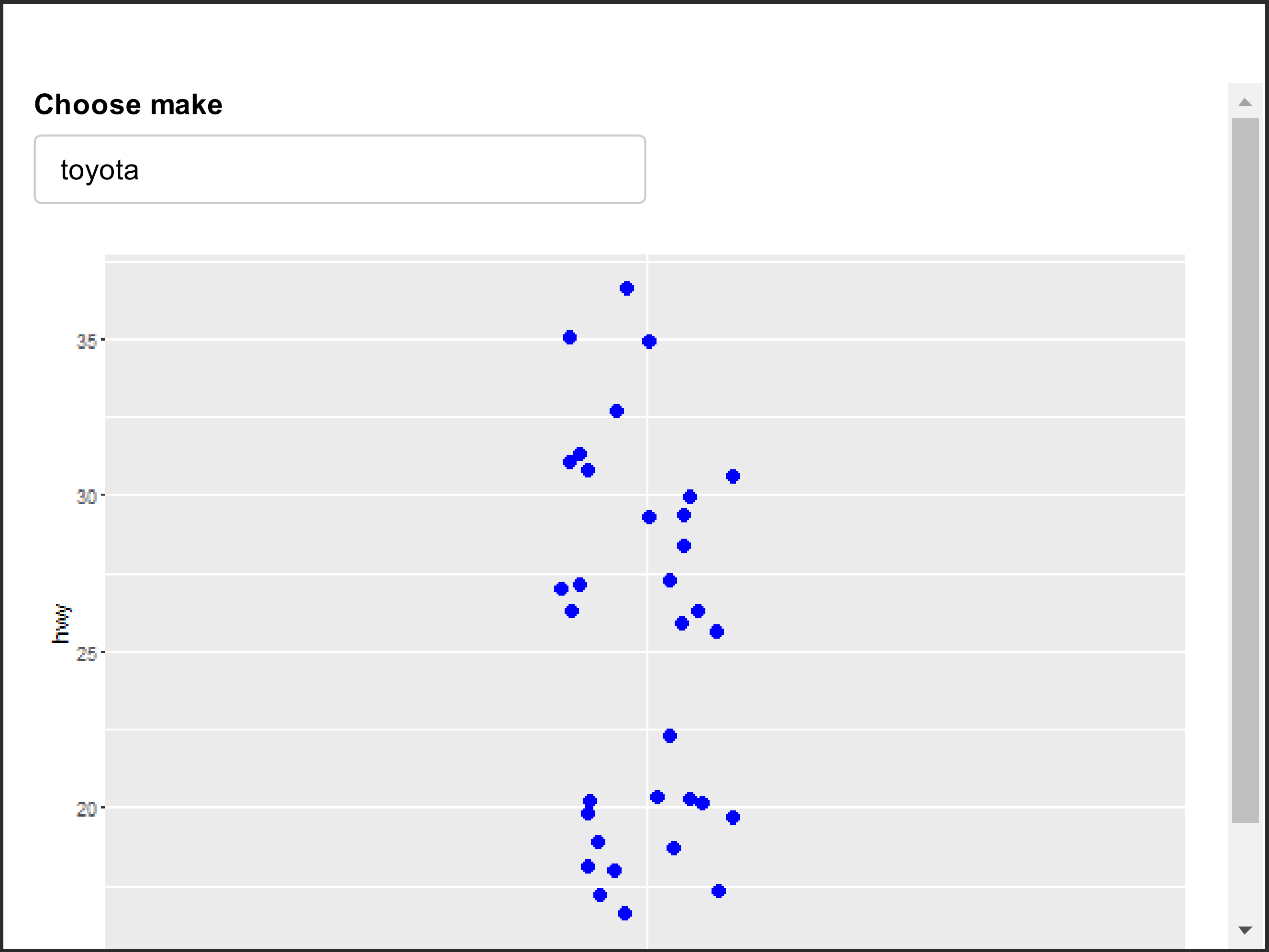
```r
# reactive to generate output
mpg2 <- reactive({mpg[mpg$manufacturer%in%input$make,]})
```

```r
# our plot renderer NOTE mpg2() with parenthesis
ouput$myplot <- renderPlot({
  ggplot(mpg2(), aes(manufacturer, hwy)) +
    geom_boxplot(color="blue")
})
```

Add manufacturer select (code)

```
library(ggplot2)
ui <- basicPage(
  selectInput("make", "Choose make",  multiple = TRUE,
choices=mpg$manufacturer, selected="toyota"),
  plotOutput("myplot")
)
server <- function(input, output, session) {

  mpg2 <- reactive({mpg[mpg$manufacturer%in%input$make,]})


  output$myplot <- renderPlot({
    ggplot(mpg2(), aes(manufacturer, hwy)) +
      geom_jitter(color="blue", width=0.2, size=3)
  })
}
shinyApp(ui = ui, server = server)
```

Add manufacturer select (app)



Choose make

toyota

Final touch, add checkbox for median sort

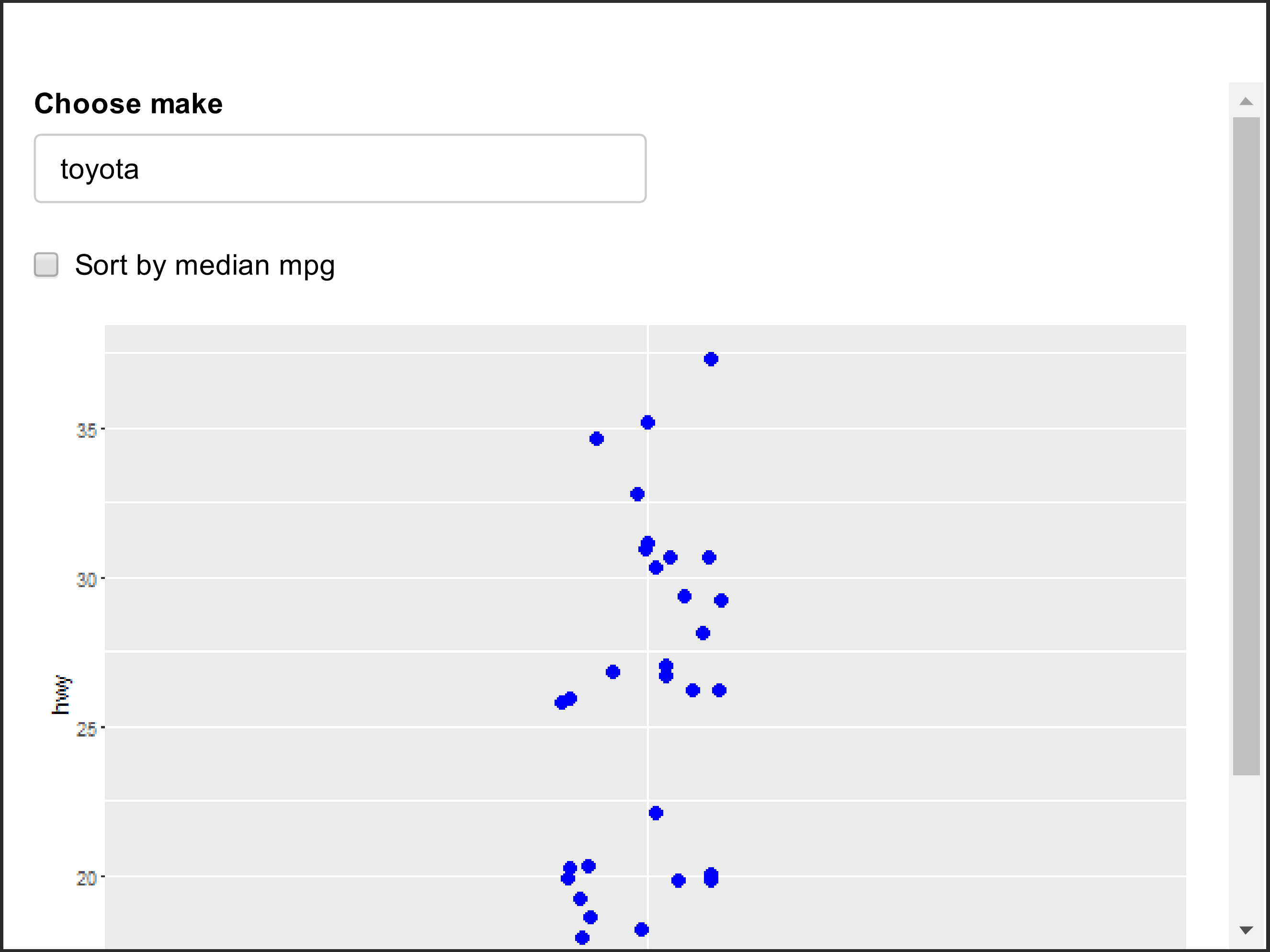
```
# changes to the UI
checkboxInput("reorder", "Sort by mpg",  FALSE),
```

```
# CHANGEs TO THE SERVER
mpg2 <- reactive({
  mpg2 <- mpg[mpg$manufacturer%in%input$make,]
  if(input$reorder) {
    mpg2$manufacturer <- reorder(mpg2$manufacturer,
mpg2$hwy, median)
  }
  return(mpg2)
})
```

Final interactive plot

```r
library(ggplot2)
ui <- basicPage(
  selectInput("make", "Choose make",  multiple = TRUE,
choices=mpg$manufacturer, selected="toyota"),
  checkboxInput("reorder", "Sort by median mpg",  FALSE),
  plotOutput("myplot")
)
server <- function(input, output, session) {

  mpg2 <- reactive({
    mpg2 <- mpg[mpg$manufacturer%in%input$make,]
    if(input$reorder) {
      mpg2$manufacturer <- reorder(mpg2$manufacturer,
mpg2$hwy, median)
    }
    return(mpg2)
  })

  output$myplot <- renderPlot({
    ggplot(mpg2(), aes(manufacturer, hwy)) +
      geom_jitter(color="blue", width=0.2, size=3)
  })
}
shinyApp(ui = ui, server = server)
```

Final interactve plot



Choose make

toyota

Sort by median mpg

Hosting/Serving Your App

- Run locally by sharing code directly or through GitHub (`runGitHub`, `runGist`)
- Use `shinyapps.io`, free for small apps
- Use Shiny Server (open source or pro)

exercise 2 (12-end, the shiny server)