

# Introduction to Shiny

Dr. Çetinkaya-Rundel

November 28, 2017

- ▶ High level view
  - ▶ Anatomy of a Shiny app
  - ▶ Reactivity 101
  - ▶ File structure
  - ▶ Sharing your app
  - ▶ Where to go next
-

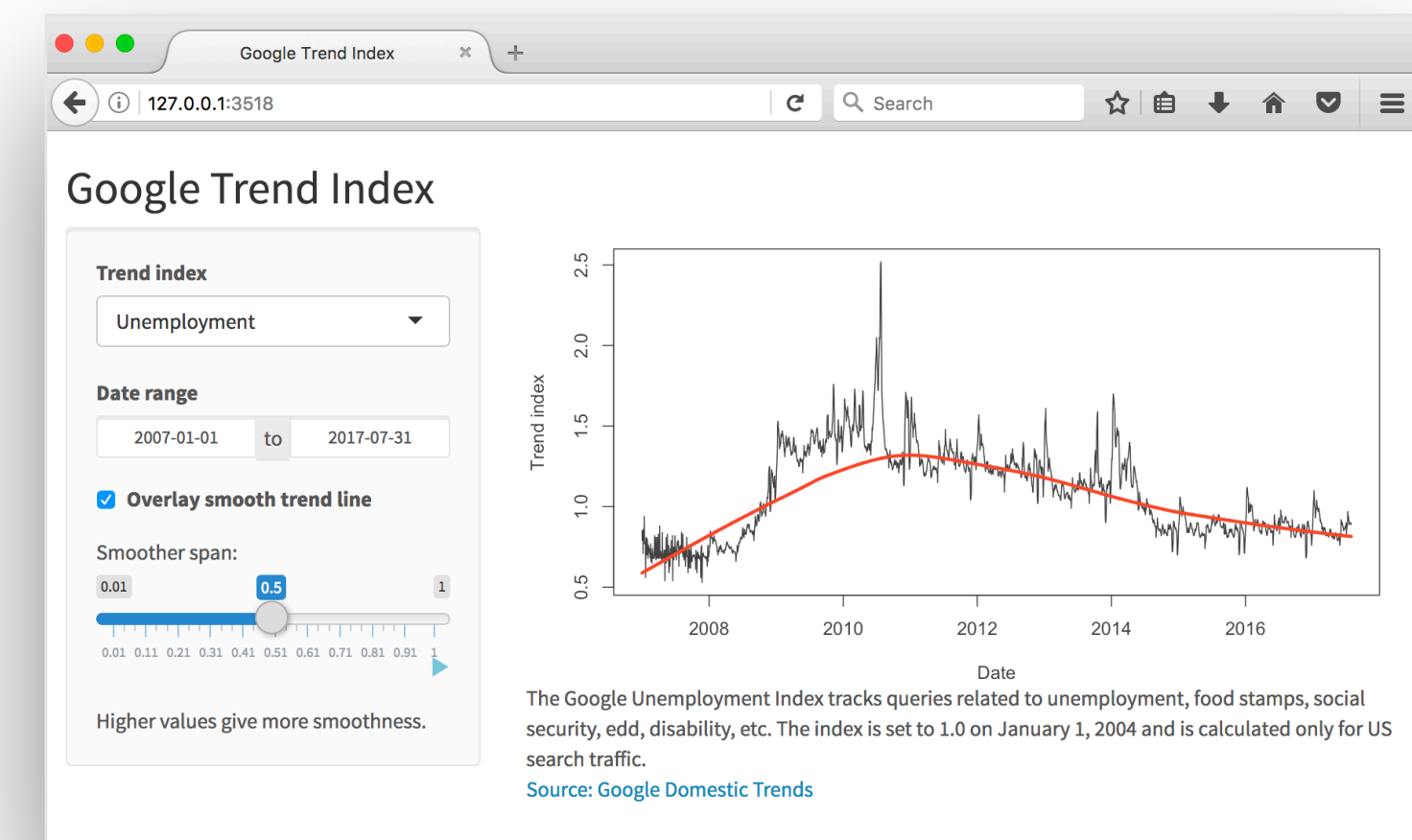


High level view

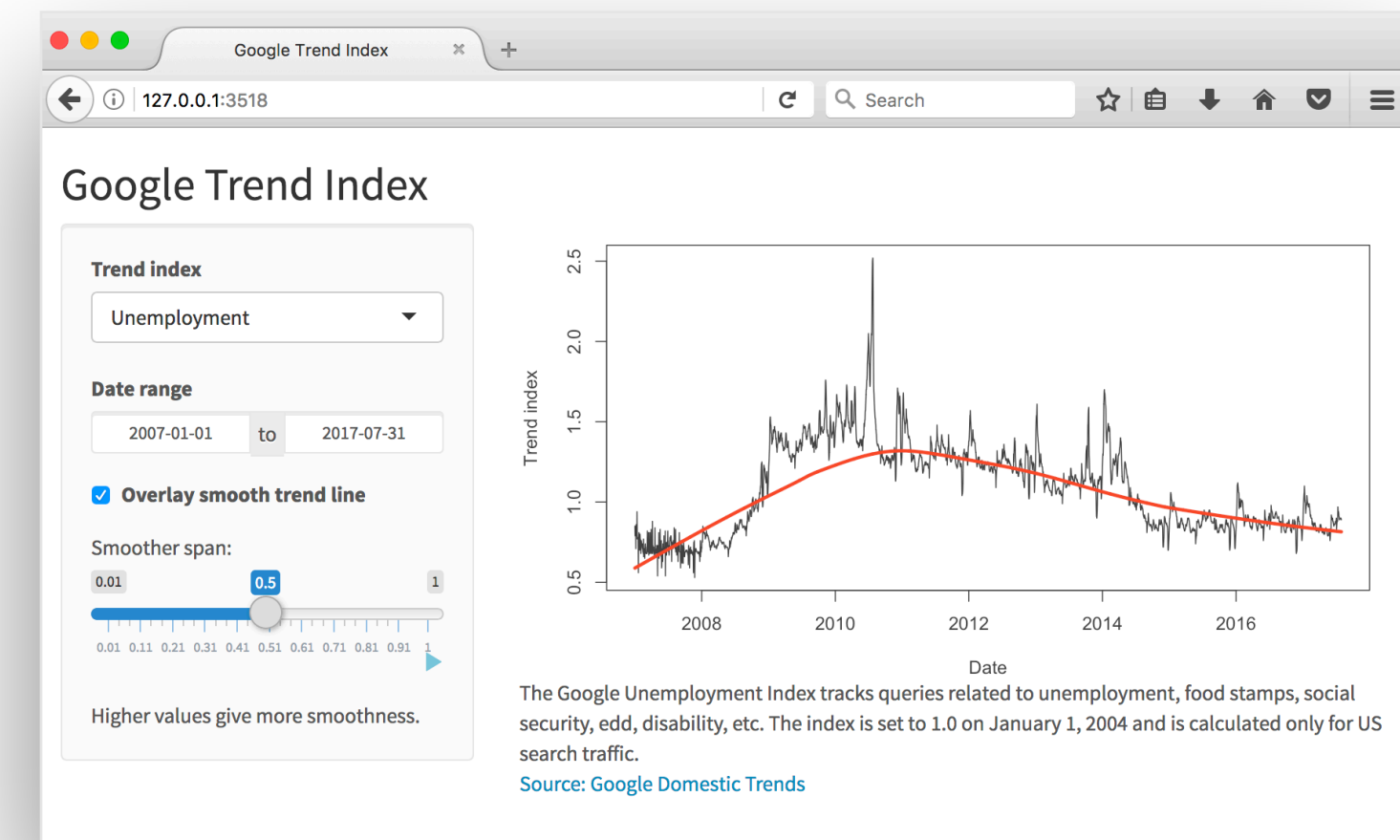
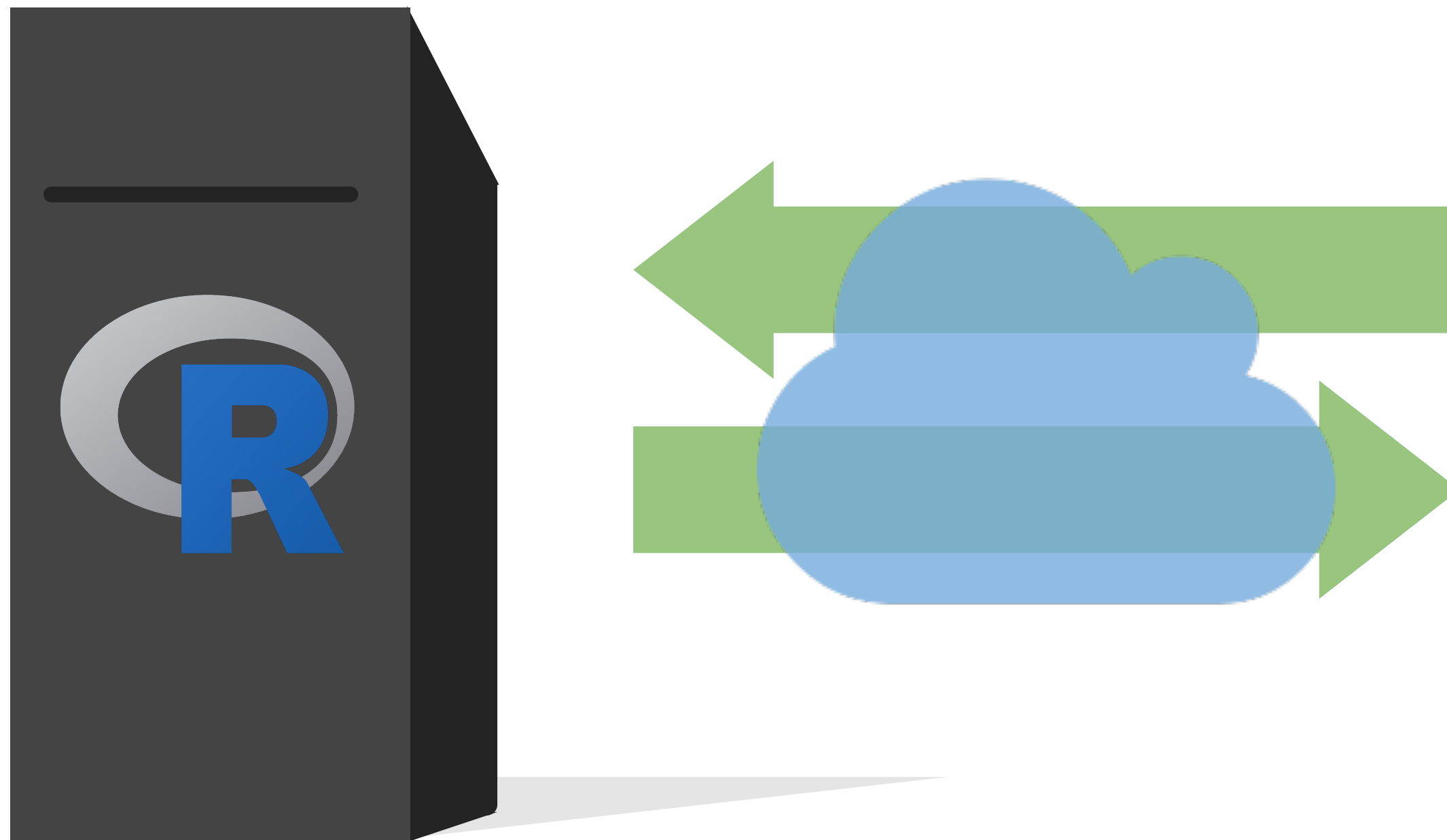
Every Shiny app has a webpage that the user visits,  
and behind this webpage there is a computer  
that serves this webpage by running R.

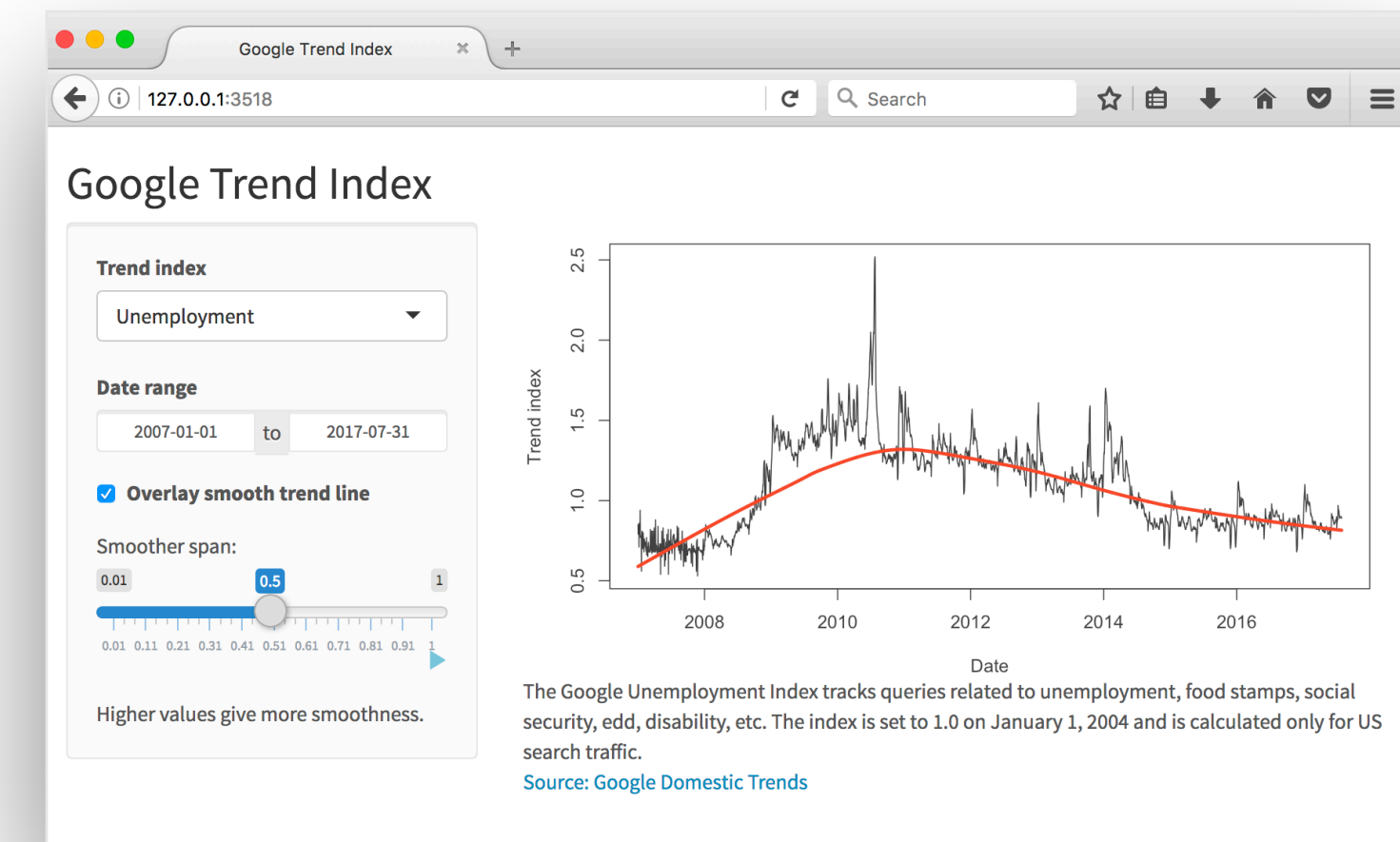
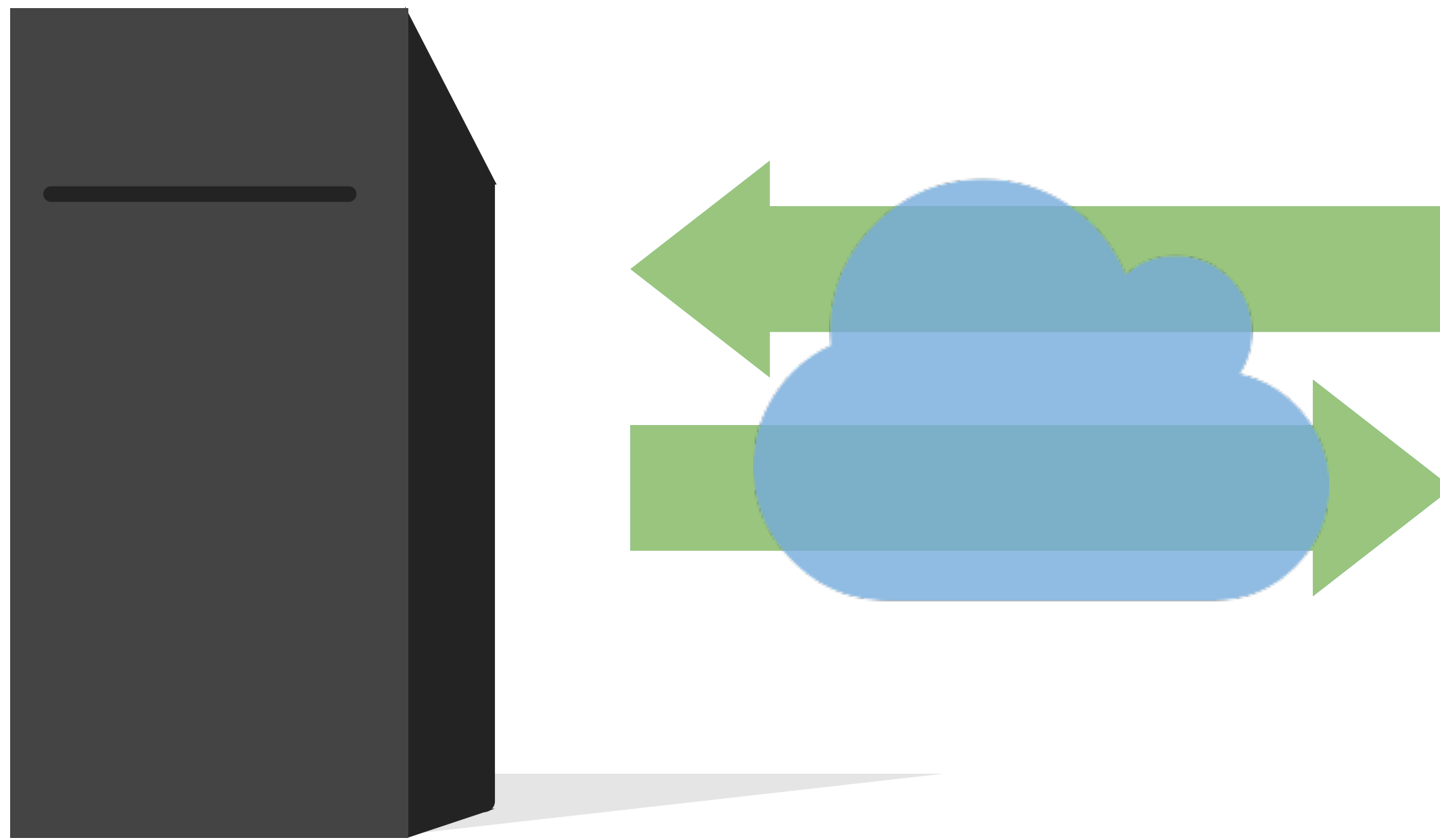


When running your app locally,  
the computer serving your app is your computer.



When your app is deployed,  
the computer serving your app is a web server.





Server instructions



User interface



apps/goog-index/app.R



# Anatomy of a Shiny app

# What's in a Shiny app?

```
library(shiny)
```

```
ui <- fluidPage()
```

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

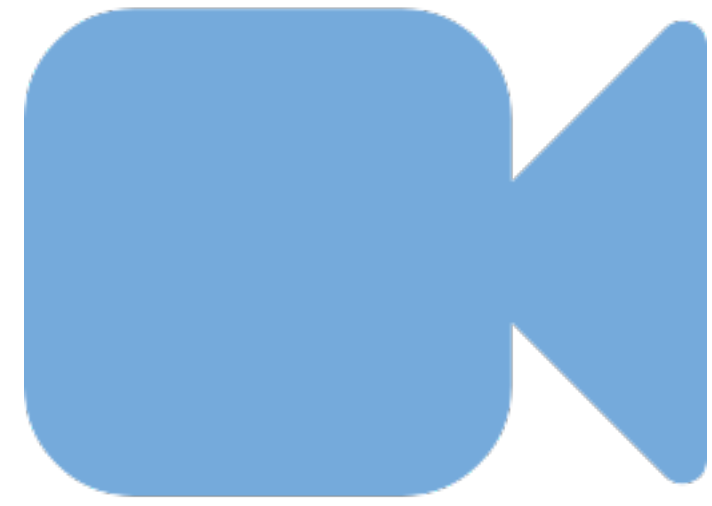
```
shinyApp(ui = ui, server = server)
```

## User interface

controls the layout and appearance of app

## Server function

contains instructions needed to build app



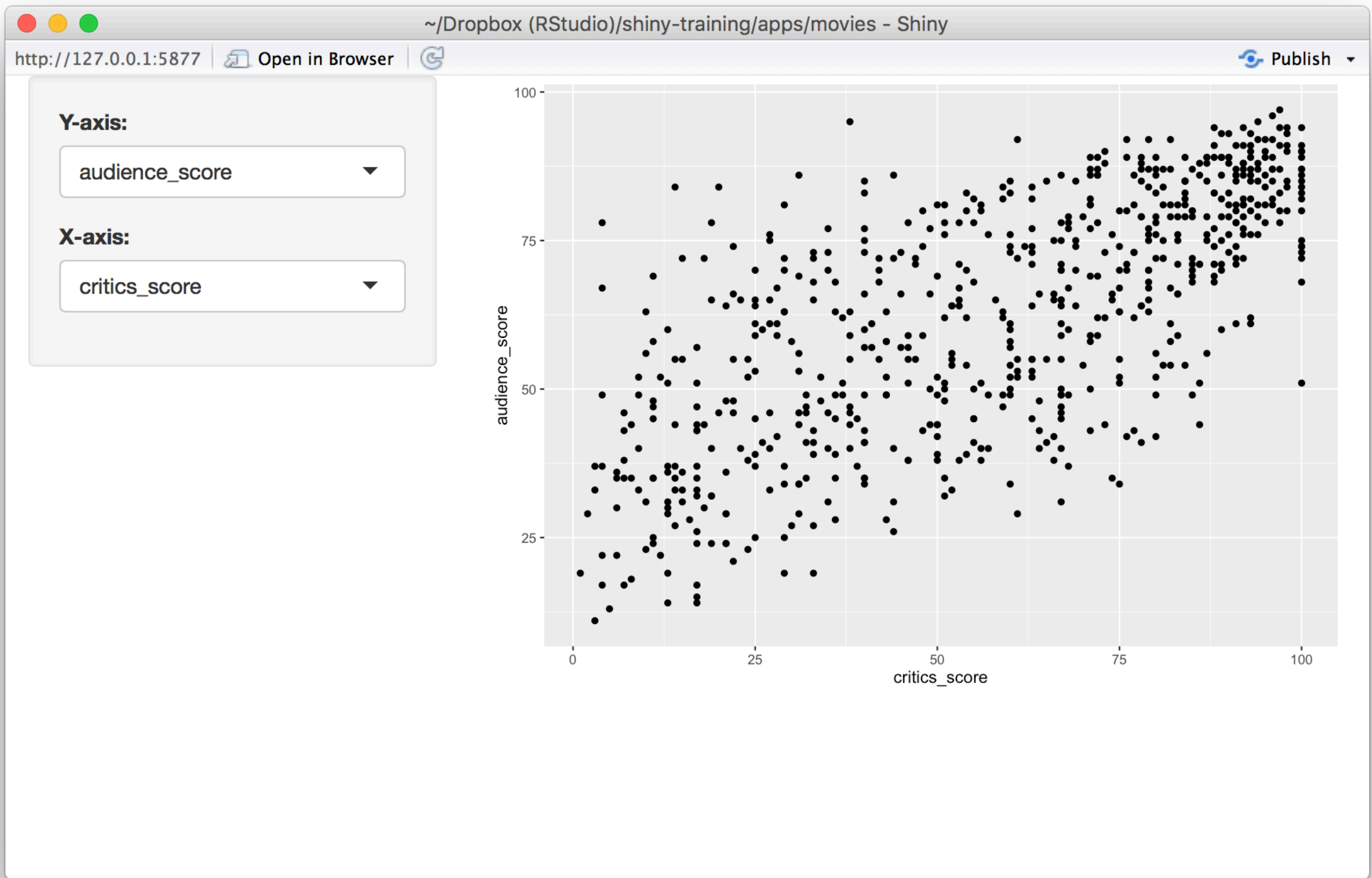
Let's build a simple movie browser app!



`data/movies.Rdata`

Data from IMDB and Rotten Tomatoes on random sample of 651 movies released in the US between 1970 and 2014

---



```
library(shiny)
library(ggplot2)
load("data/movies.Rdata")
ui <- fluidPage()

server <- function(input, output) {}

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



Dataset used for this app



```
# Define UI
ui <- fluidPage(

  # Sidebar layout with a input and output definitions
  sidebarLayout(
    # Inputs: Select variables to plot
    sidebarPanel(
      # Select variable for y-axis
      selectInput(inputId = "y", label = "Y-axis:",
                  choices = c("imdb_rating", "imdb_num_votes", "critics_score", "audience_score", "runtime"),
                  selected = "audience_score"),
      # Select variable for x-axis
      selectInput(inputId = "x", label = "X-axis:",
                  choices = c("imdb_rating", "imdb_num_votes", "critics_score", "audience_score", "runtime"),
                  selected = "critics_score")
    ),

    # Output: Show scatterplot
    mainPanel(
      plotOutput(outputId = "scatterplot")
    )
  )
)
```

```
# Define UI
```

```
ui <- fluidPage(
```

Create fluid page layout

```
# Sidebar layout with a input and output definitions
```

```
sidebarLayout(
```

```
# Inputs: Select variables to plot
```

```
sidebarPanel(
```

```
# Select variable for y-axis
```

```
selectInput(inputId = "y", label = "Y-axis:",  
             choices = c("imdb_rating", "imdb_num_votes", "critics_score", "audience_score", "runtime"),  
             selected = "audience_score"),
```

```
# Select variable for x-axis
```

```
selectInput(inputId = "x", label = "X-axis:",  
             choices = c("imdb_rating", "imdb_num_votes", "critics_score", "audience_score", "runtime"),  
             selected = "critics_score")
```

```
),
```

```
# Output: Show scatterplot
```

```
mainPanel(
```

```
  plotOutput(outputId = "scatterplot")
```

```
)
```

```
)
```



```
# Define UI
```

```
ui <- fluidPage(
```

```
# Sidebar layout with a input and output definitions
```

```
  sidebarLayout(
```

```
    # Inputs: Select variables to plot
```

```
    sidebarPanel(
```

```
      # Select variable for y-axis
```

```
      selectInput(inputId = "y", label = "Y-axis:",
```

```
                  choices = c("imdb_rating", "imdb_num_votes", "critics_score", "audience_score", "runtime"),
```

```
                  selected = "audience_score"),
```

```
      # Select variable for x-axis
```

```
      selectInput(inputId = "x", label = "X-axis:",
```

```
                  choices = c("imdb_rating", "imdb_num_votes", "critics_score", "audience_score", "runtime"),
```

```
                  selected = "critics_score")
```

```
    ),
```

```
    # Output: Show scatterplot
```

```
    mainPanel(
```

```
      plotOutput(outputId = "scatterplot")
```

```
    )
```

```
  )
```

Create a layout with a sidebar and main area

```
# Define UI
```

```
ui <- fluidPage(
```

```
# Sidebar layout with a input and output definitions
```

```
  sidebarLayout(
```

```
    # Inputs: Select variables to plot
```

```
    sidebarPanel(
```

```
      # Select variable for y-axis
```

```
      selectInput(inputId = "y", label = "Y-axis:",  
                  choices = c("imdb_rating", "imdb_num_votes", "critics_score", "audience_score", "runtime"),  
                  selected = "audience_score"),
```

```
      # Select variable for x-axis
```

```
      selectInput(inputId = "x", label = "X-axis:",  
                  choices = c("imdb_rating", "imdb_num_votes", "critics_score", "audience_score", "runtime"),  
                  selected = "critics_score")
```

```
    ),
```

```
    # Output: Show scatterplot
```

```
    mainPanel(
```

```
      plotOutput(outputId = "scatterplot")
```

```
    )
```

```
  )
```

Create a sidebar panel containing **input** controls that can in turn be passed to **sidebarLayout**

```

# Define UI
ui <- fluidPage(

  # Sidebar layout with a input and output definitions
  sidebarLayout(
    # Inputs: Select variables to plot
    sidebarPanel(
      # Select variable for y-axis
      selectInput(inputId = "y", label = "Y-axis:",
                  choices = c("imdb_rating", "imdb_num_votes", "critics_score", "audience_score", "runtime"),
                  selected = "audience_score"),
      # Select variable for x-axis
      selectInput(inputId = "x", label = "X-axis:",
                  choices = c("imdb_rating", "imdb_num_votes", "critics_score", "audience_score", "runtime"),
                  selected = "critics_score"),
    ),
    # Output: Show scatterplot
    mainPanel(
      plotOutput(outputId = "scatterplot")
    )
  )

```

**Y-axis:**

audience\_score ▼

**X-axis:**

critics\_score ▲

imdb\_rating

imdb\_num\_votes

critics\_score

audience\_score

runtime

```

# Define UI
ui <- fluidPage(

  # Sidebar layout with a input and output definitions
  sidebarLayout(
    # Inputs: Select variables to plot
    sidebarPanel(
      # Select variable for y-axis
      selectInput(inputId = "y", label = "Y-axis:",
                  choices = c("imdb_rating", "imdb_num_votes", "critics_score", "audience_score", "runtime"),
                  selected = "audience_score"),
      # Select variable for x-axis
      selectInput(inputId = "x", label = "X-axis:",
                  choices = c("imdb_rating", "imdb_num_votes", "critics_score", "audience_score", "runtime"),
                  selected = "critics_score"),
    ),

    # Output: Show scatterplot
    mainPanel(
      plotOutput(outputId = "scatterplot")
    )
  )
)

```

Create a main panel containing **output** elements that get created in the server function can in turn be passed to **sidebarLayout**



```
# Define server function
server <- function(input, output) {

  # Create the scatterplot object the plotOutput function is expecting
  output$scatterplot <- renderPlot({
    ggplot(data = movies, aes_string(x = input$x, y = input$y)) +
      geom_point()
  })
}
```

```
# Define server function
```

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

```
# Create the scatterplot object the plotOutput function is expecting
```

```
output$scatterplot <- renderPlot({
```

```
  ggplot(data = movies, aes_string(x = input$x, y = input$y)) +
```

```
    geom_point()
```

```
  })
```

```
}
```

Contains instructions  
needed to build app

```
# Define server function
```

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

```
  # Create the scatterplot object the plotOutput
```

```
  output$scatterplot <- renderPlot({  
    ggplot(data = movies, aes_string(x = input$x,  
    geom_point()  
  })
```

```
}
```

Renders a **reactive** plot that is suitable for assigning to an output slot



```
# Define server function
```

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

```
# Create the scatterplot object the plotOutput function is expecting
```

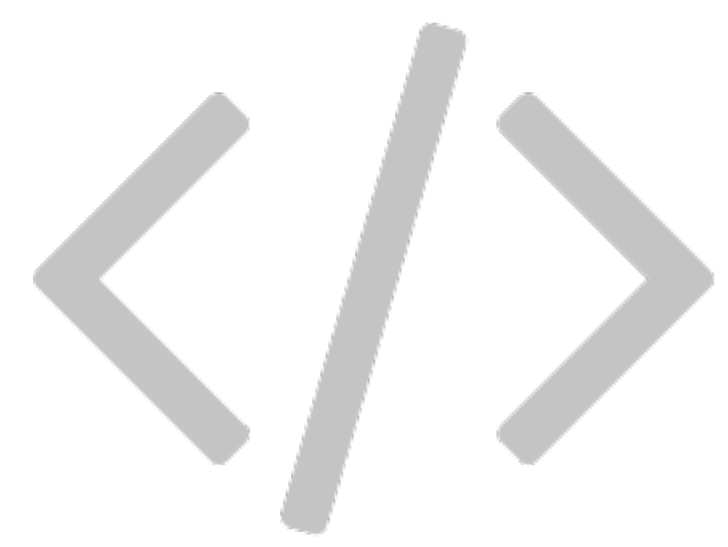
```
  output$scatterplot <- renderPlot({  
    ggplot(data = movies, aes_string(x = input$x, y = input$y)) +  
      geom_point()  
  })
```

```
}
```

Good ol' ggplot2 code,  
with **inputs** from UI



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



Putting it all together...

```
apps/movies/movies-01.R
```

---



Add a `sliderInput` for  
alpha level of points on plot

`apps/movies/movies-02.R`

[www.rstudio.com/resources/cheatsheets/](http://www.rstudio.com/resources/cheatsheets/)

## Inputs

collect values from the user

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



**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)

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

**passwordInput**(inputId, label, value)

☒ Choice A  
☐ Choice B  
☐ Choice C

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

Choice 1 | ▲

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

Choice 1  
Choice 2

0 5 10  
0 2 4 6 8 10

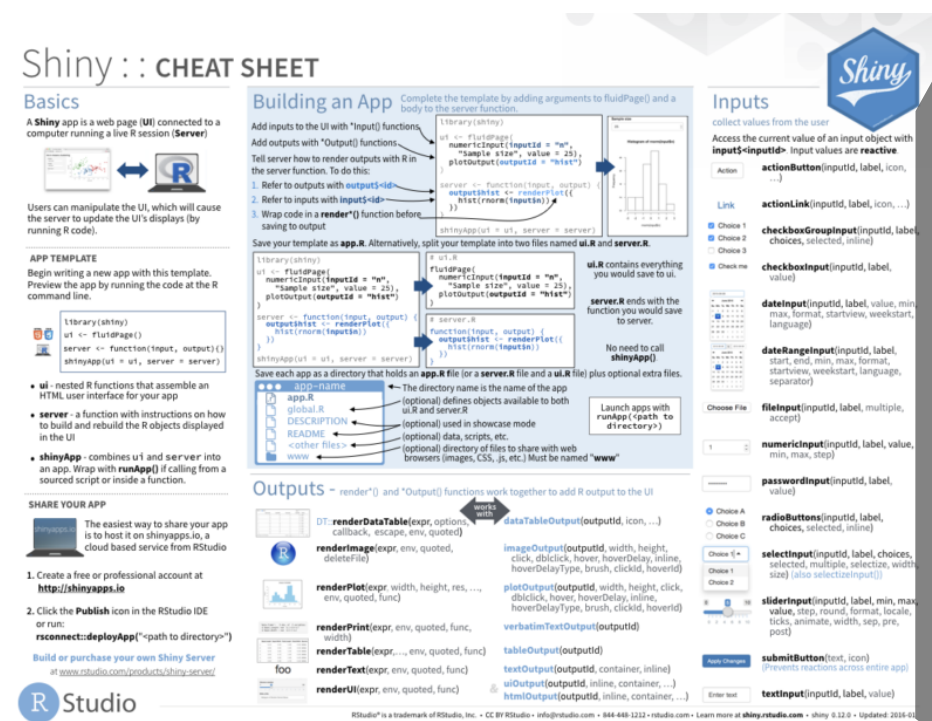
**sliderInput**(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)

Apply Changes

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

Enter text

**textInput**(inputId, label, value)

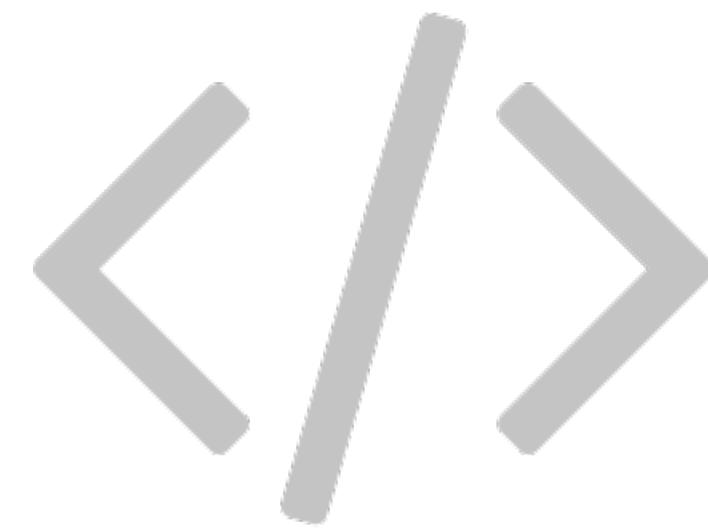




Add a new widget  
to color the points by another variable

`apps/movies/movies-03.R`

---



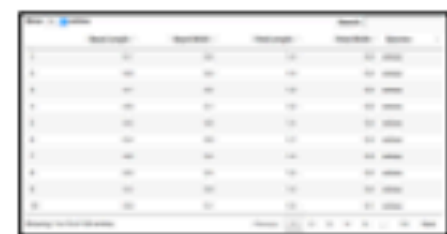
Display data frame  
*if* box is checked

`apps/movies/movies-04.R`



# Outputs

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



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

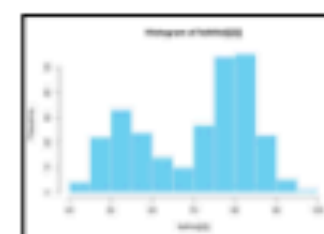
works with

**dataTableOutput**(outputId, icon, ...)



**renderImage**(expr, env, quoted, deleteFile)

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



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

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

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

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

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa

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

**renderText**(expr, env, quoted, func)

foo

**renderUI**(expr, env, quoted, func)

&

**verbatimTextOutput**(outputId)

**tableOutput**(outputId)

**textOutput**(outputId, container, inline)

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

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

Shiny :: CHEAT SHEET

Basics

- Shiny app is a web page (UI) connected to a computer running a live R session (Server)
- Users can manipulate the UI, which will cause the server to update the UI's displays by running R code.

APP TEMPLATE

Begin writing a new app with this template. Preview the app by running the code at the R command line.

• `ui` - needed R functions that assemble an HTML user interface for your app

• `server` - a function with instructions on how to build and rebuild the R objects displayed in the UI

• `shinyApp` - combines `ui` and `server` into an app. Wrap with `runApp()` if calling from a source script or inside a function.

SHARE YOUR APP

The easiest way to share your app is to host it on shinyapps.io, a cloud based service from RStudio

1. Create a free or professional account at <https://shinyapps.io>

2. Click the Publish icon in the RStudio IDE or run: `renderToShinyapps()` (path to directory)\*

\* Build or purchase your own Shiny Server

<https://www.rstudio.com/products/shiny/shiny-server/>

RStudio

Building an App

Complete the template by adding arguments to `renderUI()` and a body to the `server` function.

Add inputs to the UI with "input" functions. Add outputs with "Output" functions. Tell server how to render outputs with R in the server function. To do this:

1. Refer to outputs with `outputId`.
2. Refer to inputs with `inputId`.
3. Wrap code in a `renderUI()` function before saving to output.

Save your template as `app.R`. Alternatively, copy your template into the file named `ui.R` in `server.R`.

library(shiny)  
ui <- function() {  
 # Add UI elements here  
 # Example: text input  
 textInput("name", "Name")  
 # Example: numeric input  
 numericInput("age", "Age", value = 25)  
 # Example: select input  
 selectInput("color", "Color", choices = c("red", "green", "blue"))  
}  
server <- function(input, output, session) {  
 # Add server logic here  
 # Example: render text  
 output\$greet <- renderText({  
 paste("Hello", input\$name, "!")  
 })  
 # Example: render numeric  
 output\$age <- renderNumericInput({  
 input\$age  
 })  
 # Example: render select  
 output\$color <- renderText({  
 input\$color  
 })  
}

Shiny contains everything you would need for ui, server, and ui.R and server.R. No need to call `renderUI()`.

Launch app with `runApp()` in RStudio or `Rscript app.R` in terminal.

Optional: data, scripts, etc. (optional) directory of files to share with web browsers (images, CSS, JS, etc.) Must be named "www"

Outputs – `renderUI()` and `Output()` functions work together to add R output to the UI

`renderDataTbale`(expr, options, callback, escape, env, quoted)

`renderImage`(expr, env, quoted, deleteFile)

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

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

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

`renderText`(expr, env, quoted, func)

`renderUI`(expr, env, quoted, func)

`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, ...)

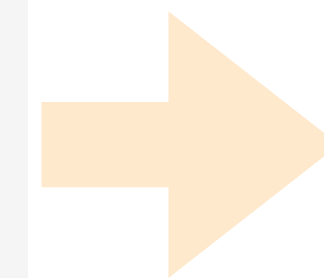
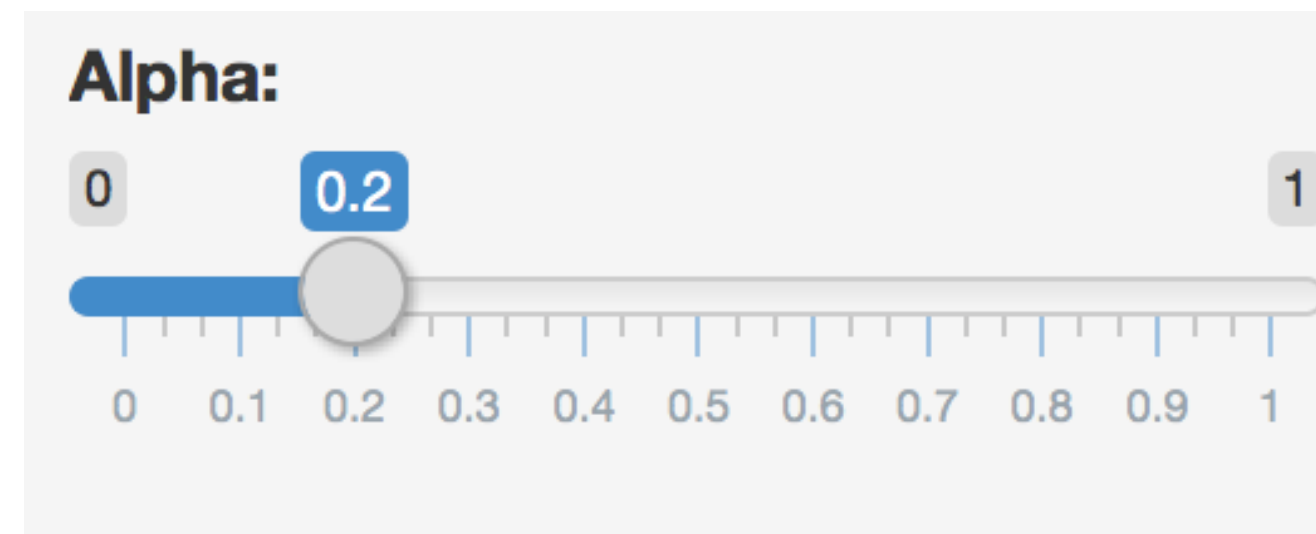
`htmlOutput`(outputId, inline, container, ...)

# Reactivity 101

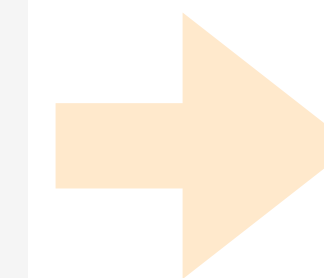
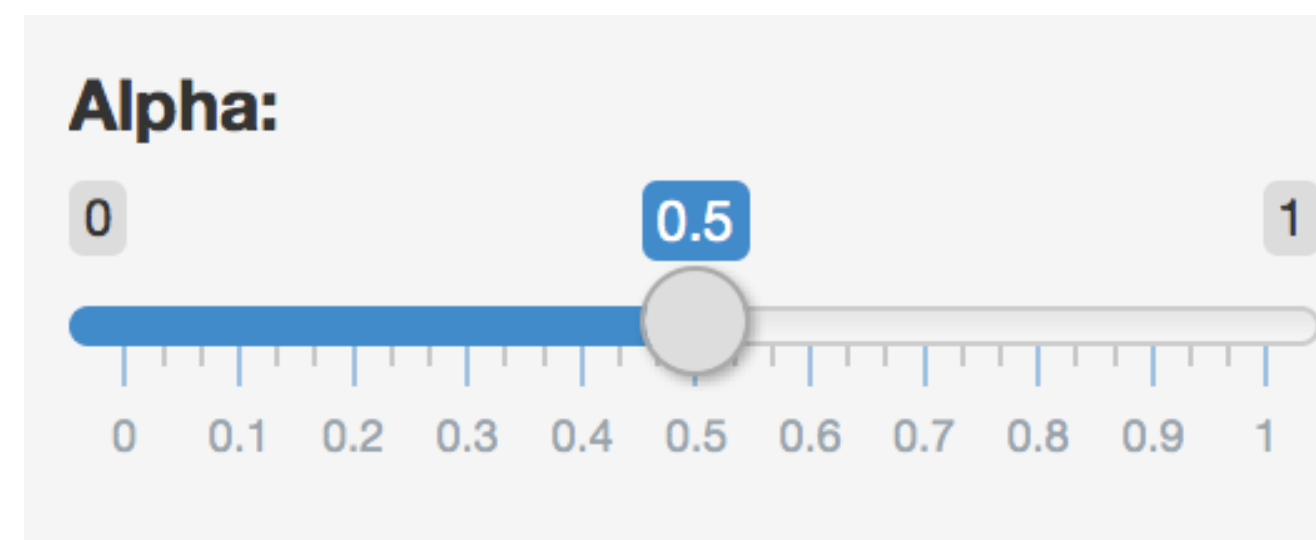
The **input\$** list stores the current value of each input object under its name.

```
# Set alpha level  
sliderInput(inputId = "alpha",  
            label = "Alpha:",  
            min = 0, max = 1,  
            value = 0.5)
```

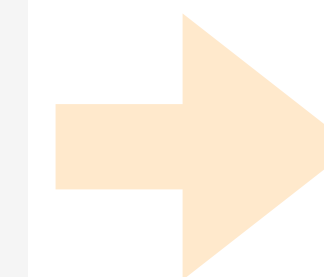
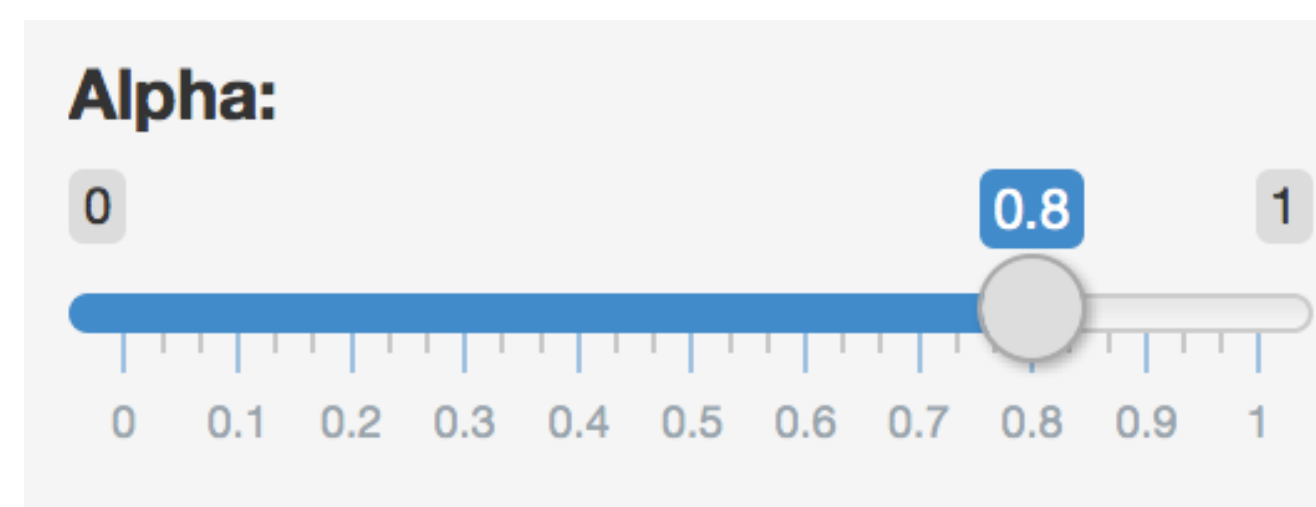
`input$alpha`



`input$alpha = 0.2`



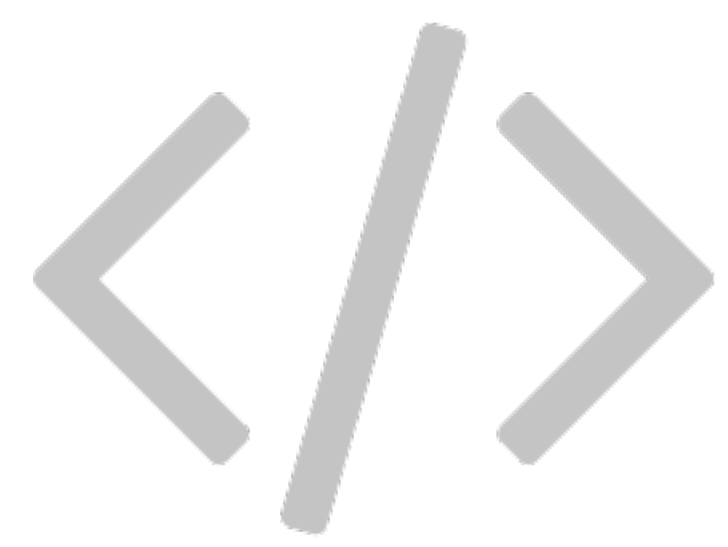
`input$alpha = 0.5`



`input$alpha = 0.8`

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

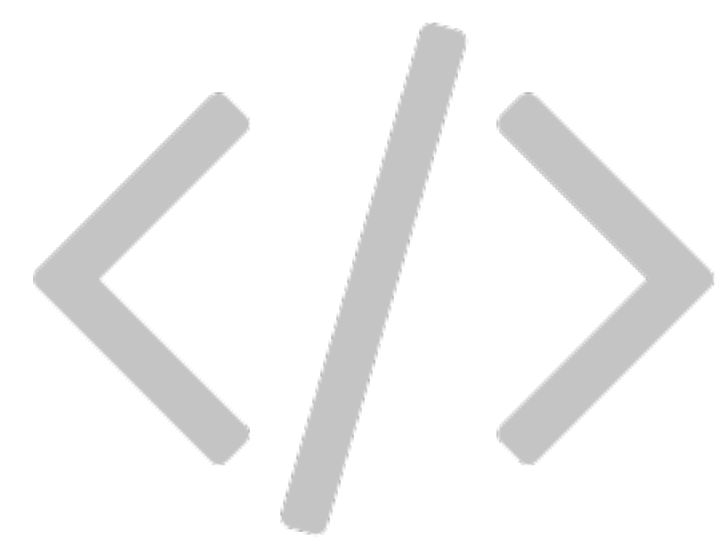
```
# Define server function required to create the scatterplot
server <- function(input, output) {
  # Create the scatterplot object the plotOutput function is expecting
  output$scatterplot <- renderPlot(
    ggplot(data = movies, aes_string(x = input$x, y = input$y,
                                     color = input$z)) +
    geom_point(alpha = input$alpha)
  )
}
```



Suppose you want the option to plot only certain types of movies as well as report how many such movies are plotted:

1. Add a UI element for the user to select which type(s) of movies they want to plot
2. Filter for chosen title type and save as a new (reactive) expression
3. Use new data frame (which is reactive) for plotting
4. Use new data frame (which is reactive) also for reporting number of observations





1. Add a UI element for the user to select which type(s) of movies they want to plot

```
# Select which types of movies to plot
checkboxGroupInput(inputId = "selected_type",
  label = "Select movie type(s):",
  choices = c("Documentary", "Feature Film",
    "TV Movie"),
  selected = "Feature Film")
```



2. Filter for chosen title type and save the new data frame as a reactive expression

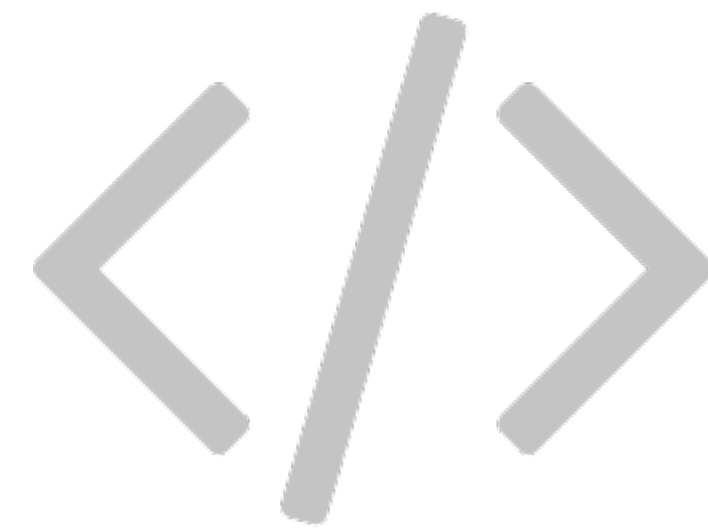
### before app:

```
library(dplyr)
```

### server:

```
# Create a subset of data filtering for  
movies_subset <- reactive({  
  req(input$selected_type)  
  filter(movies, title_type %in% input$selected_type)  
})
```

Creates a **cached expression** that knows it is out of date when input changes



### 3. Use new data frame (which is reactive) for plotting

```
# Create scatterplot object plotOutput function is expecting
output$scatterplot <- renderPlot({
  ggplot(data = movies_subset(),
    aes_string(x = input$x, y = input$y,
    geom_point(...) +
    ...
  })
```

**Cached** - only re-run  
when inputs change





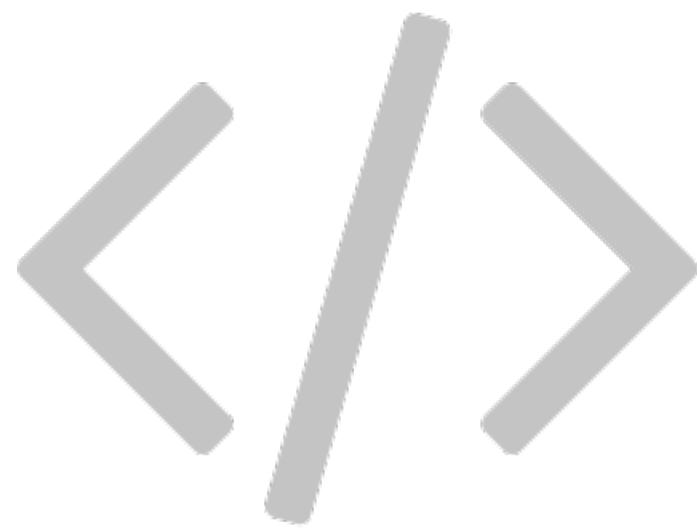
4. Use new data frame (which is reactive) also for printing number of observations

**ui:**

```
mainPanel(  
  ...  
  # Print number of obs plotted  
  uiOutput(outputId = "n"),  
  ...  
)
```

**server:**

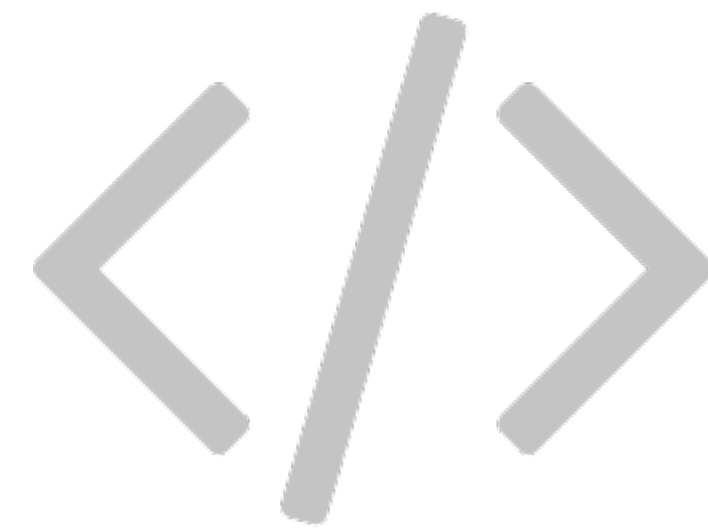
```
# Print number of movies plotted  
output$n <- renderUI({  
  types <- movies_subset()$title_type %>%  
    factor(levels = input$selected_type)  
  counts <- table(types)  
  
  HTML(paste("There are",  
             counts,  
             input$selected_type,  
             "movies in this dataset.  
             <br>"))  
})
```



Putting it all together...

```
apps/movies/movies-05.R
```

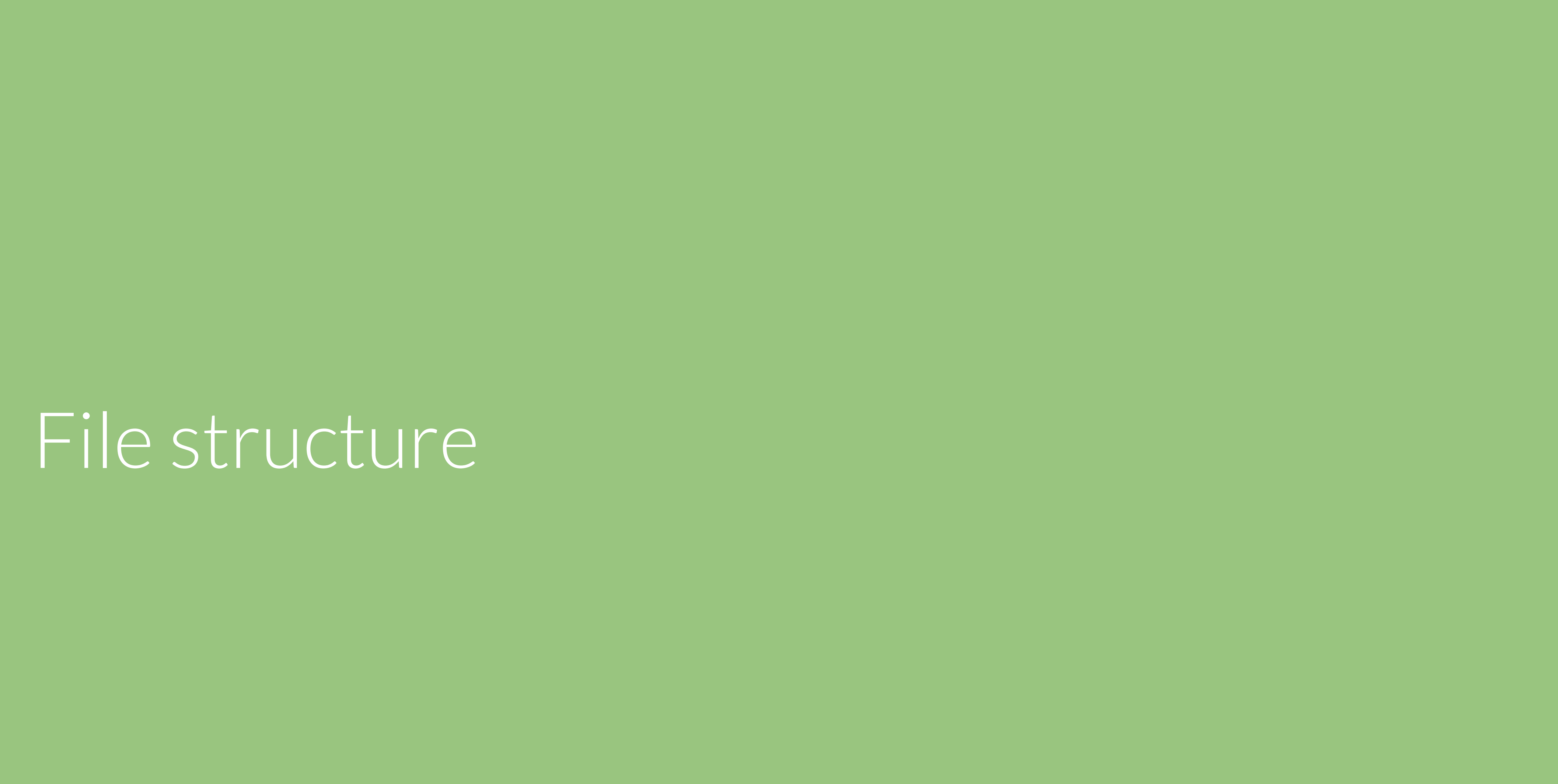
---



5. `req()`
  6. App title
  7. `selectInput()` choice labels
  8. Formatting of x and y axis labels
  9. Visual separation with horizontal lines and breaks
-

# When to use reactive

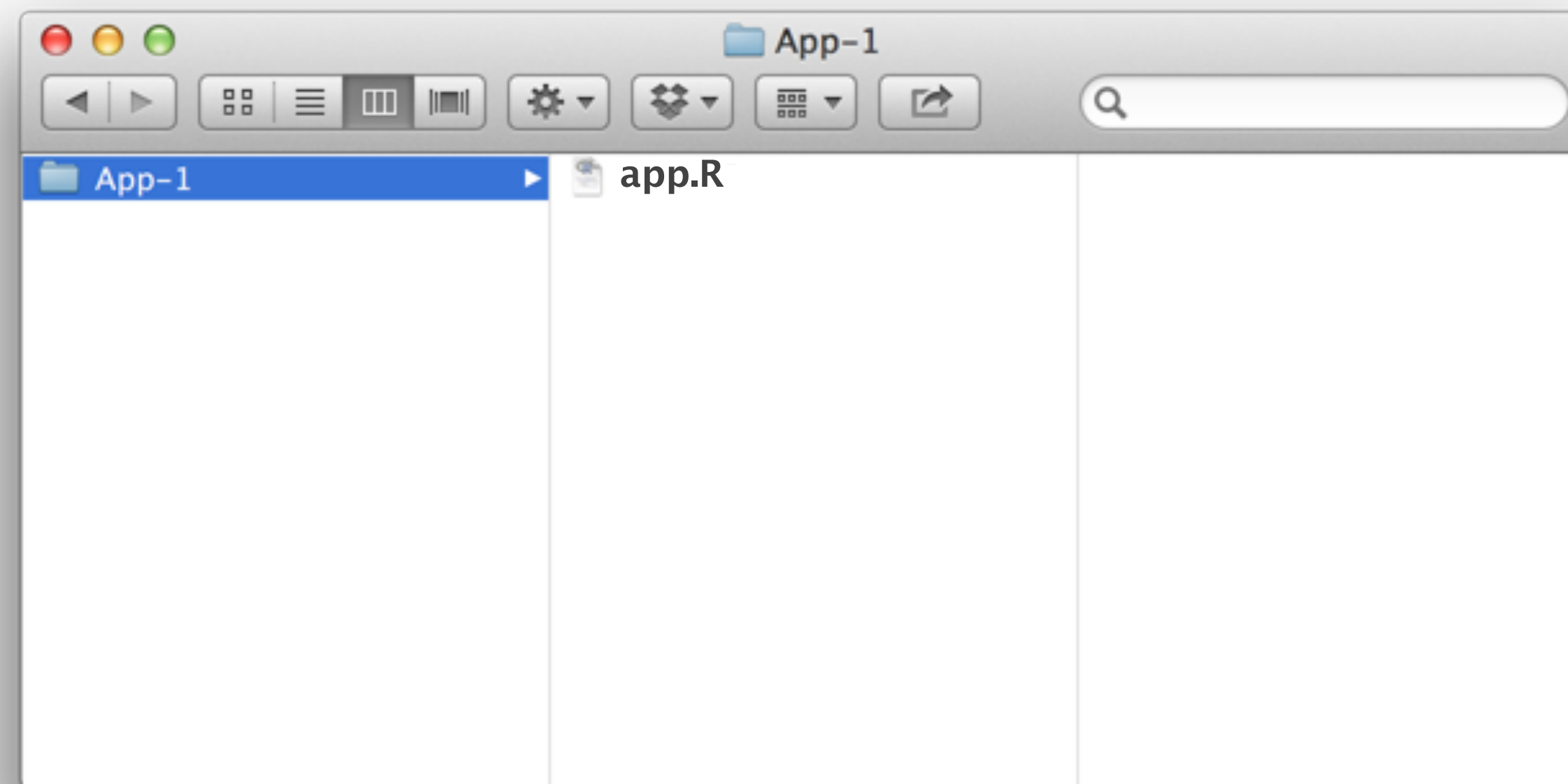
- ▶ By using a reactive expression for the subsetted data frame, we were able to get away with subsetting once and then using the result twice.
- ▶ In general, reactive conductors let you
  - ▶ not repeat yourself (i.e. avoid copy-and-paste code, which is a maintenance boon), and
  - ▶ decompose large, complex (code-wise, not necessarily CPU-wise) calculations into smaller pieces to make them more understandable.
- ▶ These benefits are similar to what happens when you decompose a large complex R script into a series of small functions that build on each other.



# File structure

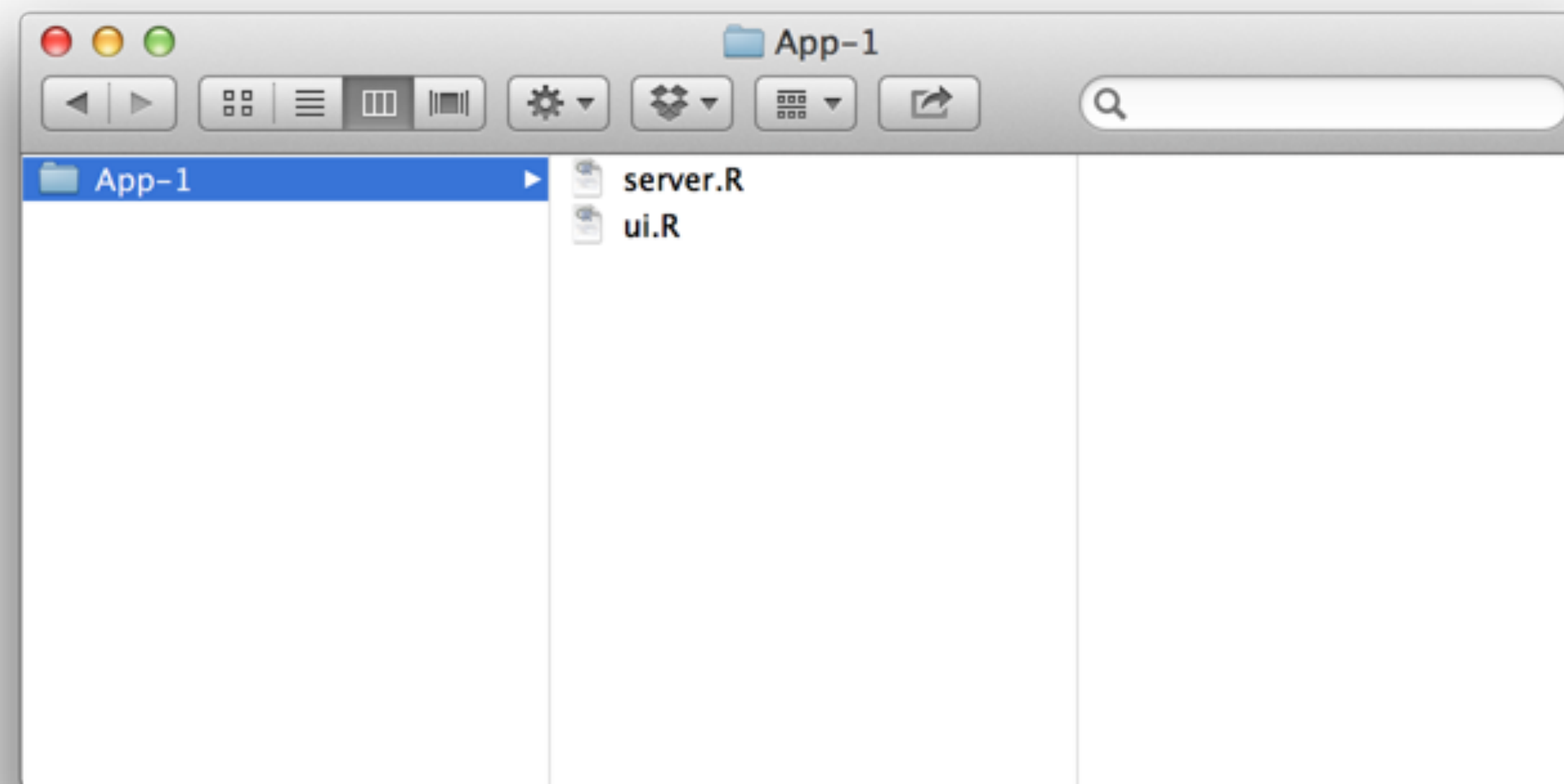
# Single file app

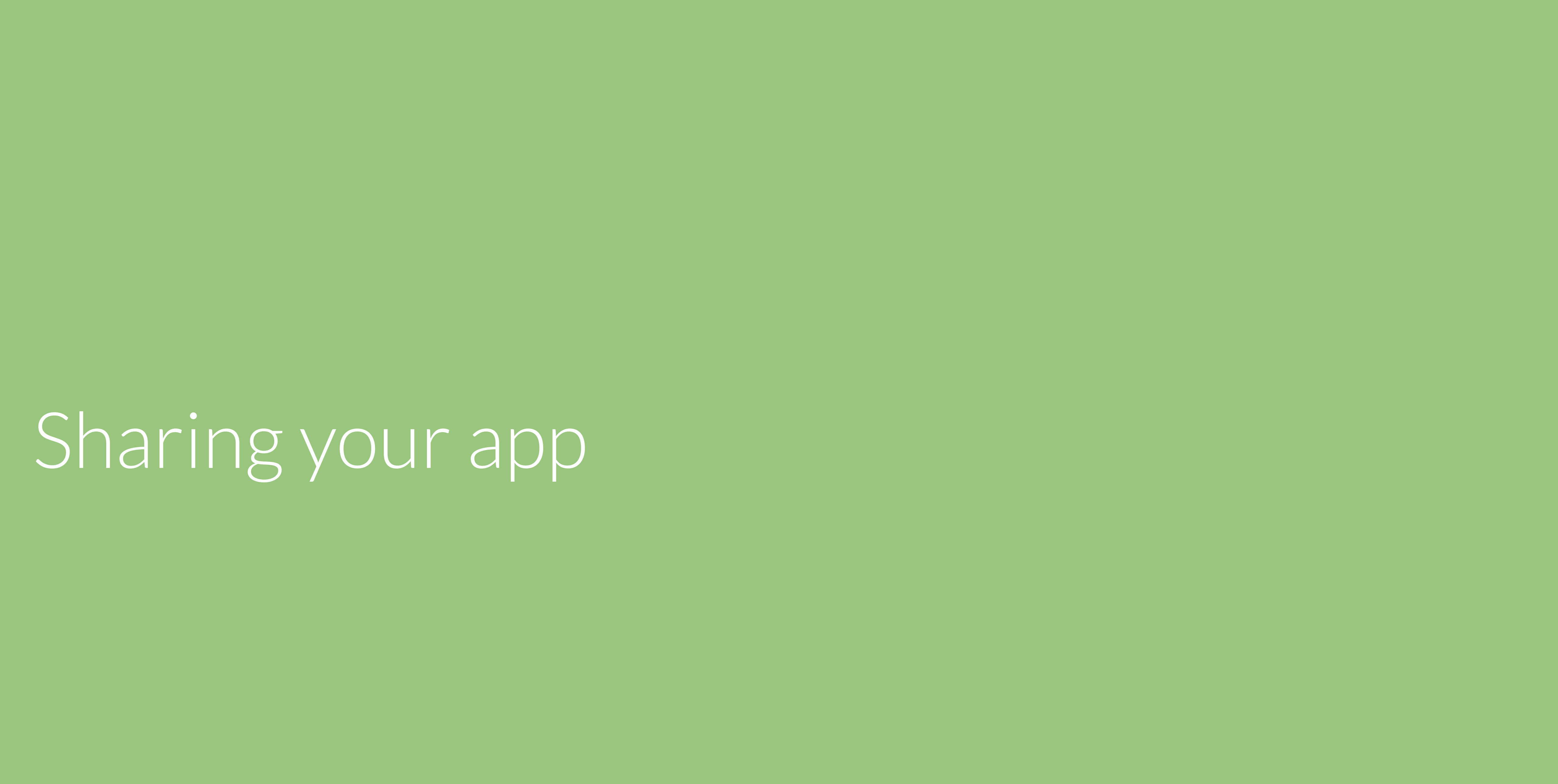
- ▶ One directory with every file the app needs:
- ▶ **app.R** (your script which ends with a call to **shinyApp()**)
- ▶ datasets, images, css, helper scripts, etc.



# Two file app

- ▶ One directory with every file the app needs:
- ▶ `server.R` and `ui.R`
- ▶ datasets, images, css, helper scripts, etc.





Sharing your app