Shiny: Introduction

Continuing Education - DIME Analytics

DIME Analytics
Development Impact Evaluation (DECDI)

Wednesday, the 4*th* of June, 2025

Session Overview 📋

Welcome to this interactive Shiny session! In the next 90 minutes, we will:

1. **Understand what Shiny is** and why it's useful
2. **Explore the structure** of a Shiny app: `UI` + `Server`
3. **Create our first Shiny app together** using the built-in template
4. Learn about **reactivity**, dynamic updates, and common **widgets**
5. **Build a multiple-file app** (`ui.R`, `server.R`, `global.R`)
6. Discover helpful **resources** and discuss your **next steps** 🚀

Let’s Do This! 🚀

This session is **live at:** 🔗 *https://ce-wb-shiny.netlify.app*

You can find the **quarto presentation** and the **final solutions** (both single-file and multiple-file apps) in our GitHub repository: 📦 *https://github.com/dime-wb-trainings/shiny-training*

What Is Shiny?

> Shiny is a web application framework for R that allows you to turn analyses into interactive web apps — all in R.

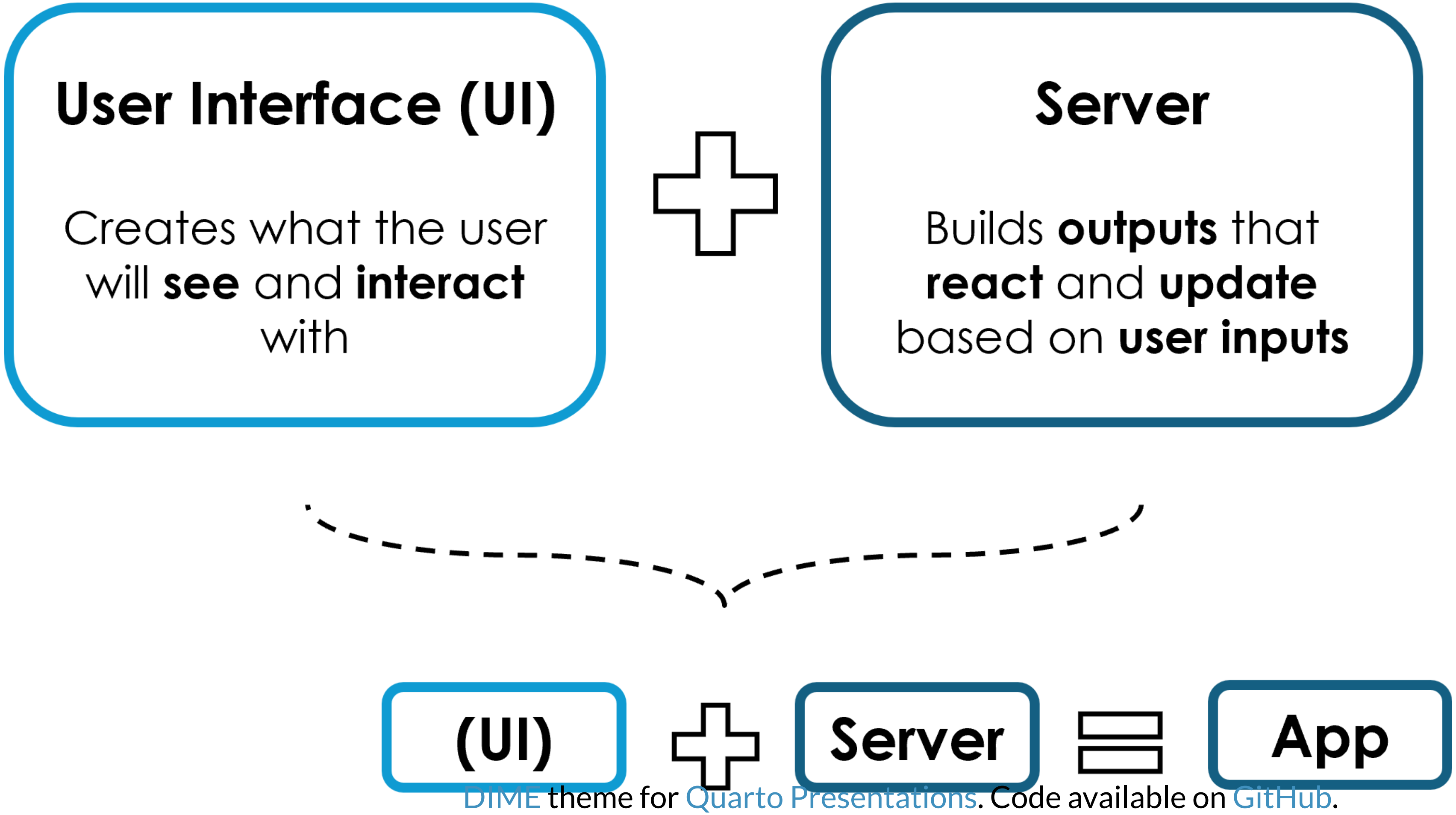
Why use it?

- Easy to learn and use
- Fast development cycle
- Powerful for data visualization
- Built on R (leverage your analysis directly)
- Great for sharing insights interactively

Anatomy of a Shiny App

A Shiny app has two core components:

- **UI (User Interface)**: Defines how the app looks
- **Server**: Defines how the app works


User Interface (UI)
Creates what the user will **see** and **interact** with
Server
Builds **outputs** that **react** and **update** based on **user inputs**
(UI)
Server
App

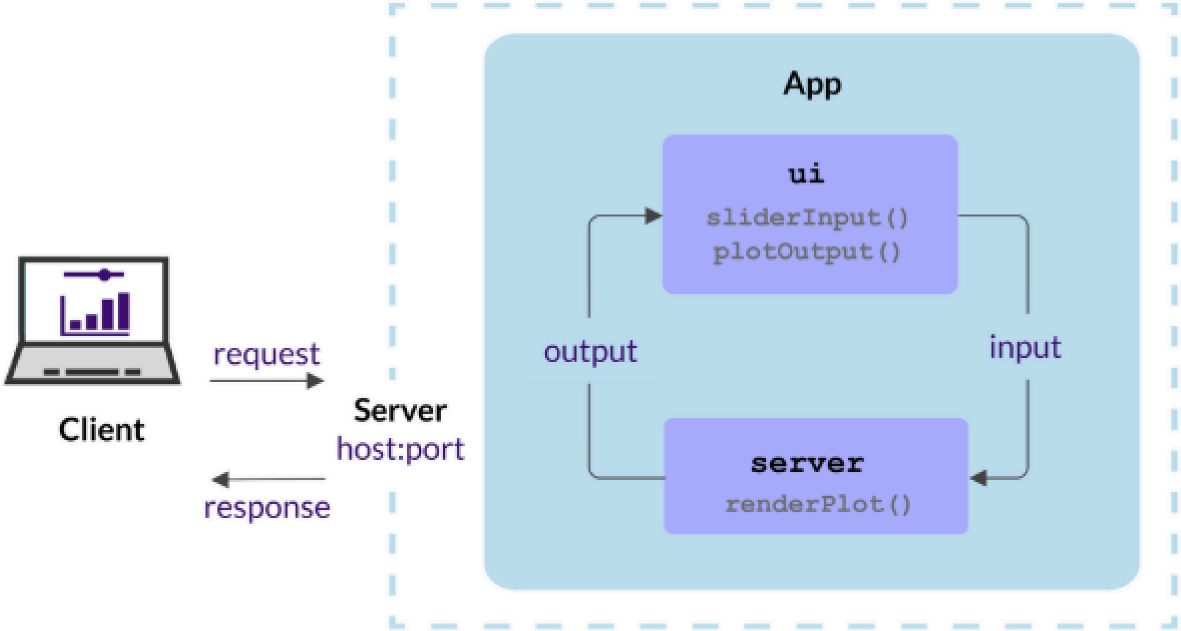
The Client, Host, and Server

Apps are served to users via a host and port. The R session running the server reacts to user actions, computes results, and sends them back.

Client: The web browser where the user interacts with the app

Host:Port: Shiny app is served at an IP (host) and port

Server: Runs R session to monitor inputs and respond with outputs



DIME theme for Quarto Presentations. Code available on GitHub.

Let’s Build Our First App Together (with the help of the R template) ✍️

Step-by-step instructions:

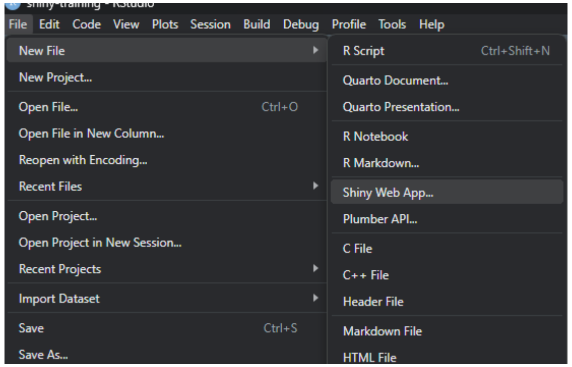
1. Open RStudio
2. If you haven’t already installed Shiny, run:

```r
install.packages("shiny")
```

3. Load the Shiny library:

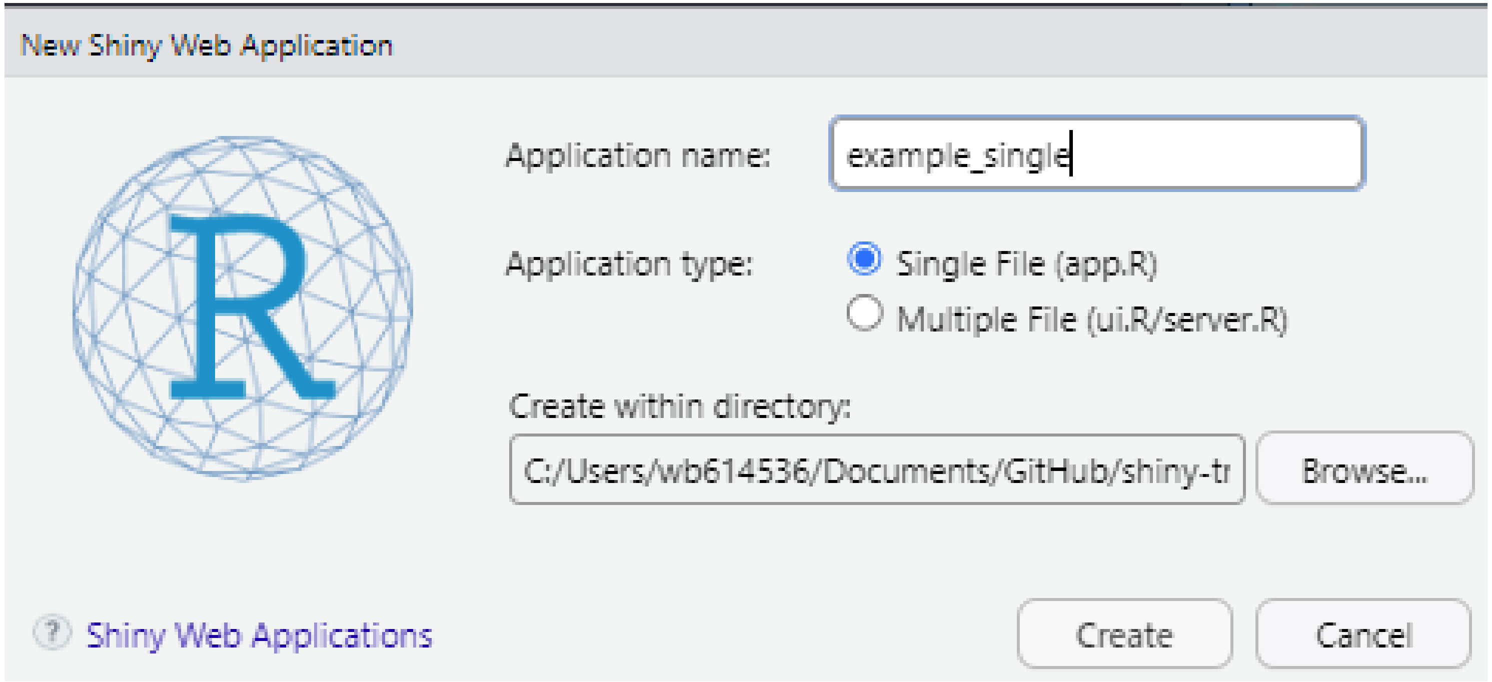
```r
library(shiny)
```

4. Create a new Shiny Web App: Click on **File > New File > R Shiny Web App**



Let’s Build Our First App Together (with the help of the R template) ✍️

5. Choose **Single File** option when prompted:



New Shiny Web Application

Application name: example_single

Application type: ● Single File (app.R) ○ Multiple File (ui.R/server.R)

Create within directory: C:/Users/wb614536/Documents/GitHub/shiny-tr

6. Name your folder and click OK

7. Click **Run App** in the top-right corner

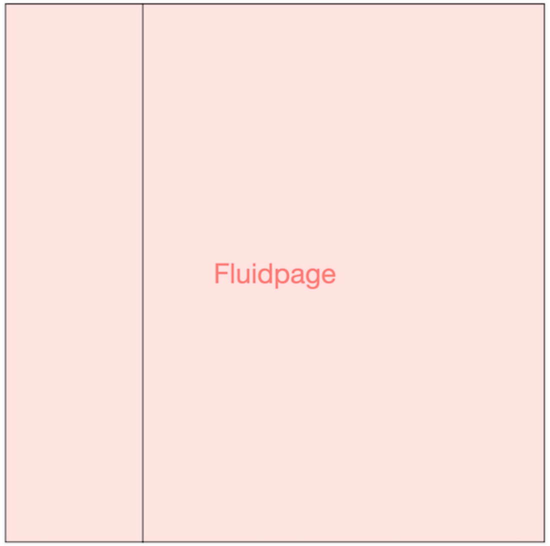
8. 🎉 You’re running your first Shiny app!

Let’s Explore the Code 🧠

> Go to the app you just created and let’s explore each element

1. `ui` — User Interface

```r
ui <- fluidPage( #<<
  titlePanel("Old Faithful Geyser Data"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  )
)
```



Layout elements

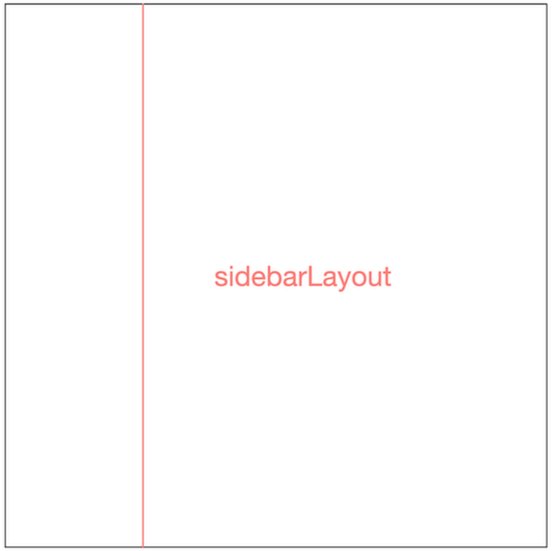
- `fluidPage()` is the container for the app interface, the layout in which your content is. This is the most common, but there are other types of layouts. Check here

Let’s Explore the Code 🧠

> Go to the app you just created and let’s explore each element

1. `ui` — User Interface

```r
ui <- fluidPage(
  titlePanel("Old Faithful Geyser Data"),
  sidebarLayout( #<<
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  )
)
```



Layout elements

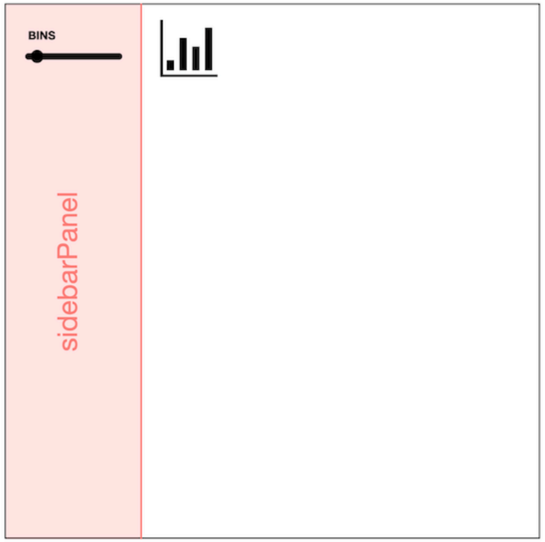
- **`sidebarLayout()`** splits the layout into sidebar (sidebarPanel()) and main area (mainPanel()). This is also optional.

Let’s Explore the Code 🧠

> Go to the app you just created and let’s explore each element

1. `ui` — User Interface

```r
ui <- fluidPage(
  titlePanel("Old Faithful Geyser Data"),
  sidebarLayout(
    sidebarPanel( #<<
      sliderInput("bins",  #<<
                  "Number of bins:",  #<<
                  min = 1, #<<
                  max = 50, #<<
                  value = 30) #<<
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  )
)
```



Page content

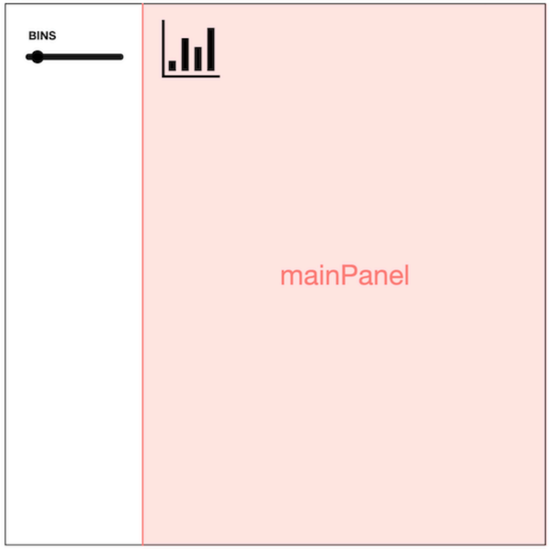
- `sidebarPanel()` contains one input field (`sliderInput()`) called “bins”. This is a slider that lets a user choose a number. As you can infer from the name, in this case, it’s the number of histogram bins.

Let’s Explore the Code 🧠

> Go to the app you just created and let’s explore each element

1. `ui` — User Interface

```r
ui <- fluidPage(
  titlePanel("Old Faithful Geyser Data"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),
    mainPanel( #<<
      plotOutput("distPlot") #<<
    )
  )
)
```



Page content

- `mainPanel()` contains a histogram (`plotOutput()`), which will be defined in the server function. The name, or id, of this histogram is **“distPlot”**

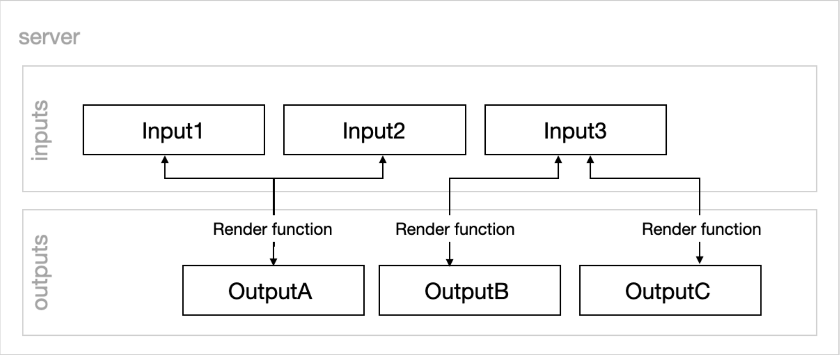
Shiny Server Logic Explained ⚙️

2. `server` — Server Logic

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

The `server()` function takes two arguments:

- `input`: a **reactive list** of input values from the UI
- `output`: a **reactive list** where you assign render functions



Reactive lists are “special” lists used in **reactive programming** — a way to make your app **automatically update** outputs when inputs change.

Shiny Server Logic Explained ⚙️

2. `server` — Server Logic

```r
server <- function(input, output) {
  output$outputId <- renderFunction({
    value <- input$inputId
})
}
```

Let’s take a look at reactivity inside a simple `server( )`:

- **`renderFunction`**: A function like `renderPlot()`, `renderTable()`, etc. used to render an output (a plot, a table…)
- **`outputId`**: Identifies the rendered output in the output list (`output$`) for the UI
- The function regenerates **`value`** every time the input field referenced by **inputID** in the **input** list changes.

Shiny Server Logic Explained ⚙️

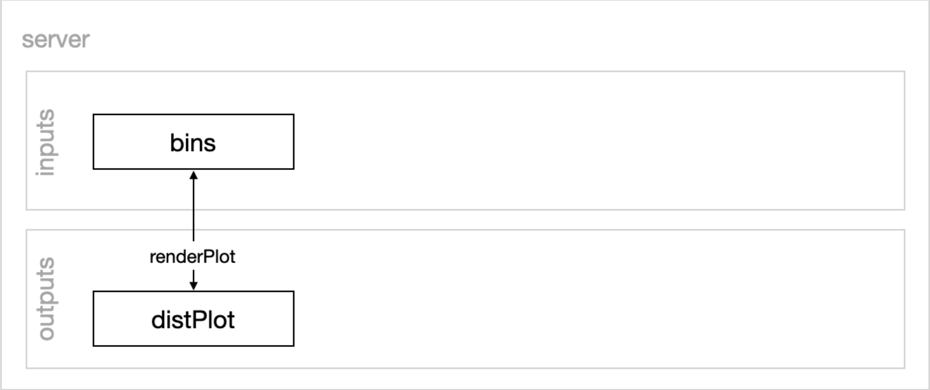
2. `server` — Server Logic

```r
server <- function(input, output) {
  output$distPlot <- renderPlot({
    x <- faithful[, 2]
    bins <- seq(min(x), max(x), length.out = input$bins + 1)

    hist(x, breaks = bins, col = 'darkgray', border = 'white',
         xlab = 'Waiting time to next eruption (in mins)',
         main = 'Histogram of waiting times')
  })
}
```

In our case:

- the server contains the logic to create the histogram `distPlot` in the output list (`output$`), using the render function `renderPlot()`.
- `distPlot` depends on one user input (`input$bins`), which pulls the number from the slider input in the UI.



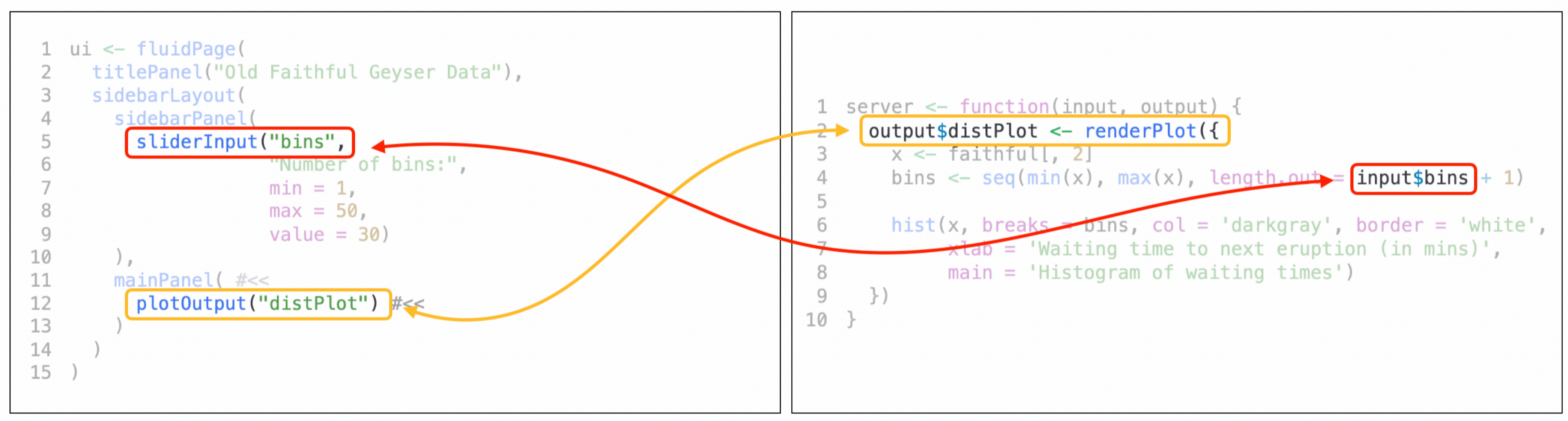
➡️ **Result**: the histogram updates as the slider moves!

Shiny Server Logic Explained ⚙️

How Server Connects to UI 🧩

Remember these connections?

UI	Server
`plotOutput("distPlot")`	`output$distPlot <- renderPlot()`
`sliderInput("bins", ...)`	`input$bins`



```r
ui <- fluidPage(
  titlePanel("Old Faithful Geyser Data"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),
    mainPanel( #<<
      plotOutput("distPlot") #<<
    )
  )
)
```

```r
server <- function(input, output) {
  output$distPlot <- renderPlot({
    x <- faithful[, 2]
    bins <- seq(min(x), max(x), length.out = input$bins + 1)

    hist(x, breaks = bins, col = 'darkgray', border = 'white',
         xlab = 'Waiting time to next eruption (in mins)',
         main = 'Histogram of waiting times')
  })
}
```

DIME theme for Quarto Presentations. Code available on GitHub.

Inputs & Outputs in Shiny 🧠

Shiny apps are built by connecting **inputs** (from the UI) to **outputs** (rendered in the server).

Part	Role	Examples
`ui`	Define layout and inputs/outputs	`sliderInput()`, `plotOutput()`
`server`	Logic to render outputs based on inputs	`renderPlot()`, `renderText()`

🔁 **Reactivity** connects them:

- `input$...` pulls values from UI controls
- `output$... <- render...()` generates dynamic content

A recap on Reactivity 🔁

- **Reactive programming** lets your app respond to changes without needing to re-run code manually.
- `input` and `output` behave like **reactive lists** — not regular R lists, but special objects in Shiny.
- When a user selects a value on the slider, say **5**, Shiny stores it as `input$bins = 5`.
- If the user changes it to **7**, Shiny automatically updates the `input` list — and any **render function** using it will re-execute.
- This is why `output$distPlot <- renderPlot({ ... input$bins ... })` updates **instantly**.

Together, `input`, `output`, and `render*()` functions form the **reactive backbone** of your app.

Common Input Widgets 🛠

Shiny includes many built-in **widgets** to capture user input:

Widget	Purpose	Example Use
`numericInput()`	Enter a number	Age, price
`sliderInput()`	Select from a range	Histogram bins
`selectInput()`	Choose from a list	Country selector
`radioButtons()`	Choose one option	Plot type
`textInput()`	Enter text	Comments, filters
`fileInput()`	Upload a file	CSV, Excel
`actionButton()`	Trigger an action manually	Run, Submit

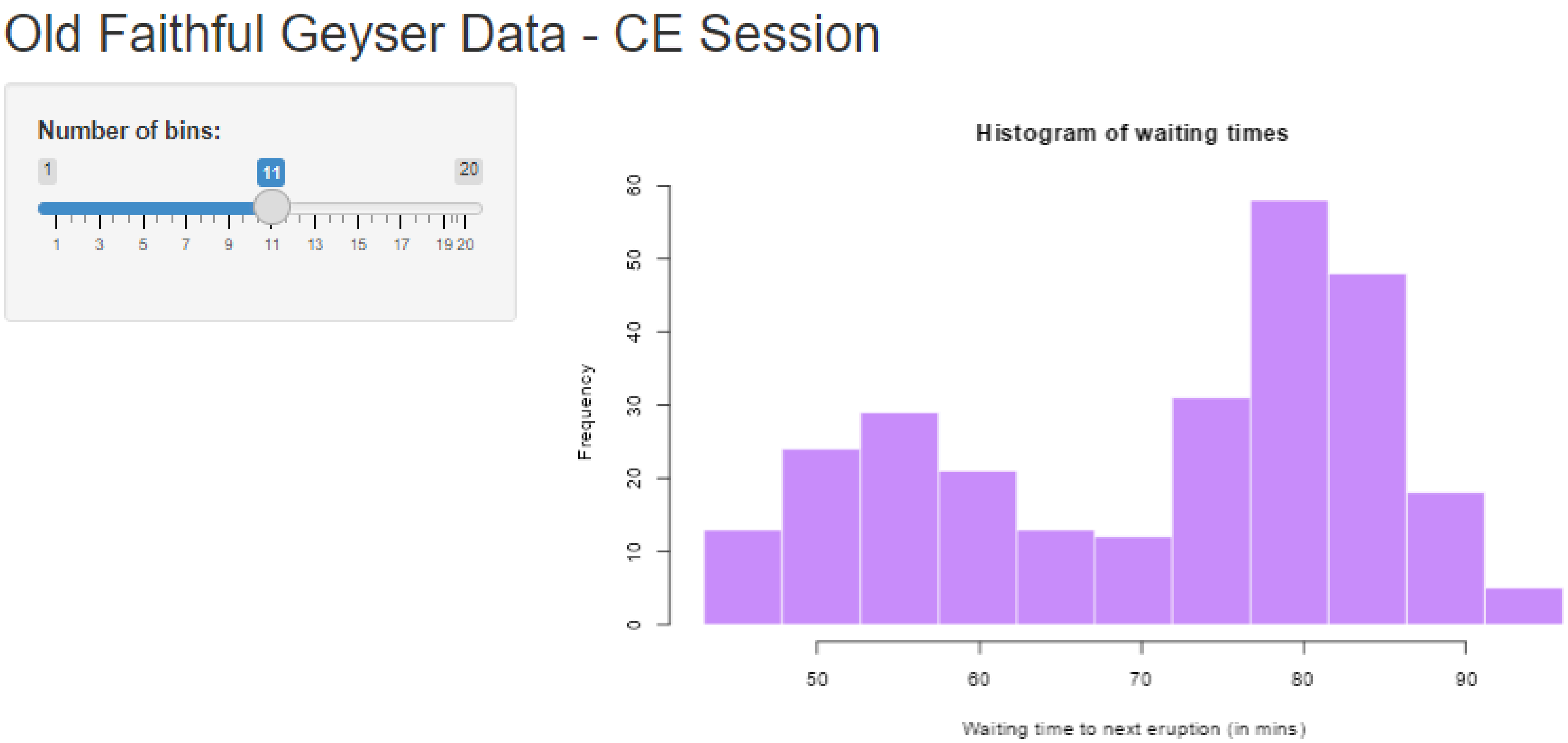
📚 See the full gallery: ➡️ Shiny Widgets Gallery

Let’s start with some basic modifications ✍️

1. Change the title of the app
2. Change the number of bins to 20
3. Change the color of the histogram to `#ca8dfd` (a shade of purple)

Let’s start with some basic modifications ✍️

After your modifications the app should look like this:



Old Faithful Geyser Data - CE Session

Number of bins: 11

1
20
1
3
5
7
9
11
13
15
17
19
20
Histogram of waiting times
Frequency
0
10
20
30
40
50
60
50
60
70
80
90
Waiting time to next eruption (in mins)

- The app with the modifications is available here

Behind the Scenes: Running a Shiny App ⚙️

Before you close the app, check the **R console**. You'll see something like:

```
#> Listening on http://127.0.0.1:3827
```

🔍 **What it means:** - `127.0.0.1` refers to your local machine ("localhost") - `3827` is a random port number - You can open the app in any browser using this address

⛔ **While the app is running:** - The R console is blocked (no new commands allowed) - A stop sign appears in the RStudio toolbar

🛑 **To stop the app:** - Click the stop sign icon - Press `Esc` (or `Ctrl + C` in terminal) - Close the Shiny app window

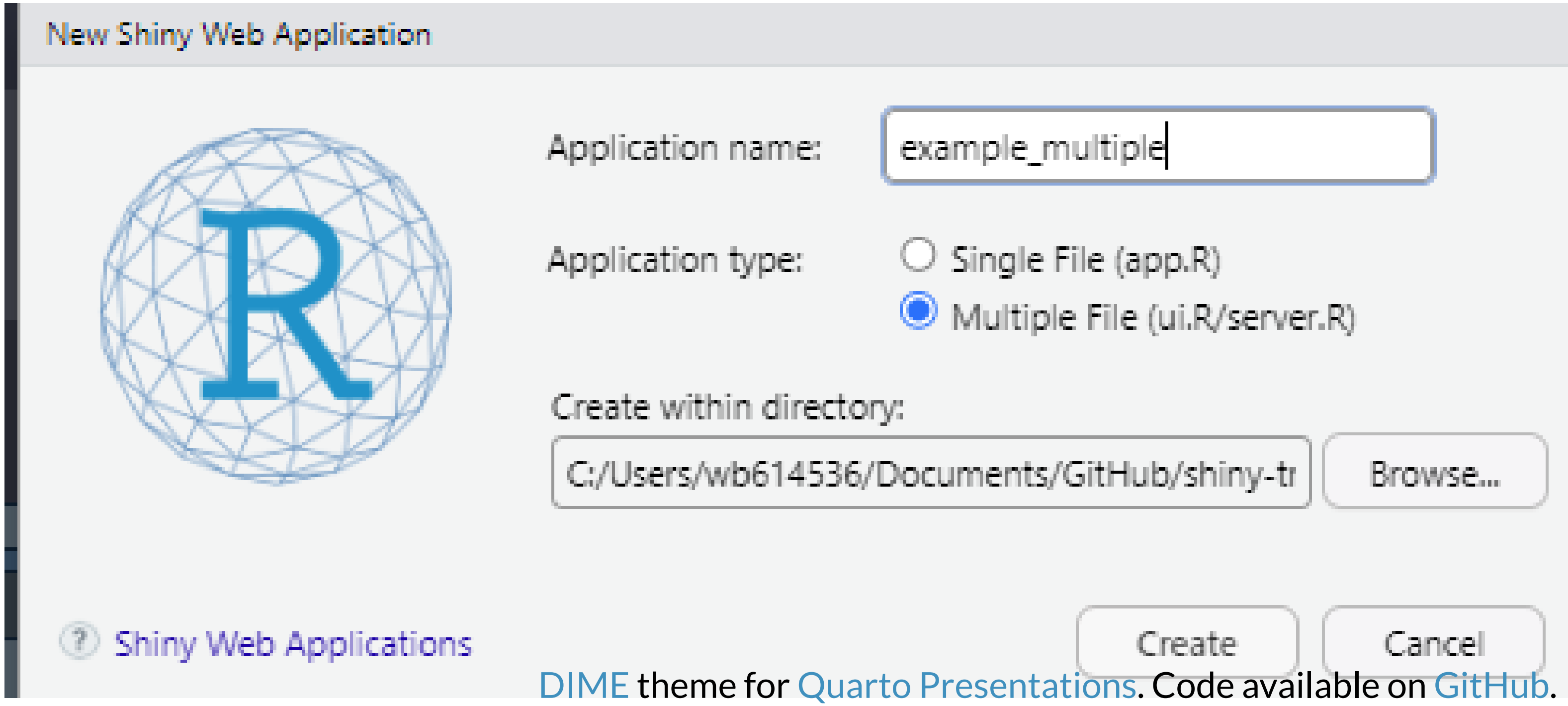
Questions

Building a Multiple -File App 🏗️

As your app grows, managing everything in a **single file** becomes difficult. That's why it's a good idea to switch to a **multi-file structure** — this is the recommended approach.

Let's walk through how to set it up!

1. In RStudio, go to
   `File > New File > Shiny Web App…`
2. This time, choose "**Multiple File**" when prompted:



New Shiny Web Application

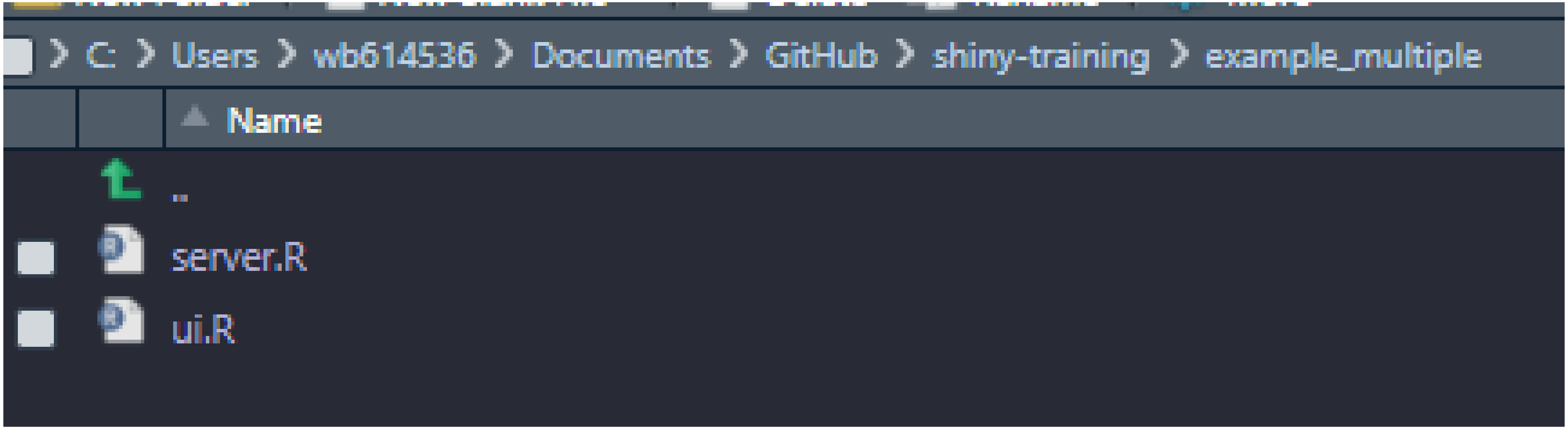
Application name: example_multiple

Application type:
○ Single File (app.R)
● Multiple File (ui.R/server.R)

Create within directory: C:/Users/wb614536/Documents/GitHub/shiny-tr

Building a Multiple -File App 🏗️

3. Name your project folder and click OK. This will automatically create two files.



C: > Users > wb614536 > Documents > GitHub > shiny-training > example_multiple

4. Lastly let’s create an extra file `global.R`. (Optional but recommended) This file is useful for loading packages and defining global objects or functions used by both `ui.R` and `server.R`.

5. Click the **Run App** button in the top-right corner of RStudio.

✨ Now Let's Make It More Interesting

- You've set up a multiple-file **Shiny app**—great start! Now let's **customize it together**.
- We'll go through a series of **hands-on exercises** using the `faithful` dataset to:
  - Add new **UI components**
  - Enhance **server logic**
- After each exercise, we'll do a **live walkthrough** to see how the changes integrate into the app.

📝 **Note:** While we're using the files created by RStudio as a starting point, you're not limited to that setup. You can always:

- Create a Shiny app by saving your own `.R` scripts as `ui.R` and `server.R`
- Or combine everything into a single `app.R` file if you prefer that style

Exercise 1: Add a Custom Title and Subtitle ✍️

Let’s improve the layout and presentation of your app!

📝 Your task:

- In `ui.R`, replace the `titlePanel()` with:
  - A custom **title**
  - A smaller **subtitle** using `h3()`

🔍 Hint:

```r
titlePanel("Faithful Geyser Data - Customized"),
h3("Exploring waiting times between eruptions")
```

Exercise 2: Add a Color Selector ✍️

Let’s make the histogram more interactive!

📝 Your task:

- Add a `selectInput()` to the `sidebarPanel()` so users can choose a color for the histogram
- Then use `input$color` inside `renderPlot()` to apply the color.

🔍 Hint:

UI:

```r
selectInput("color", "Choose a color:", choices = c("turquoise", "plum", "orchid"))
```

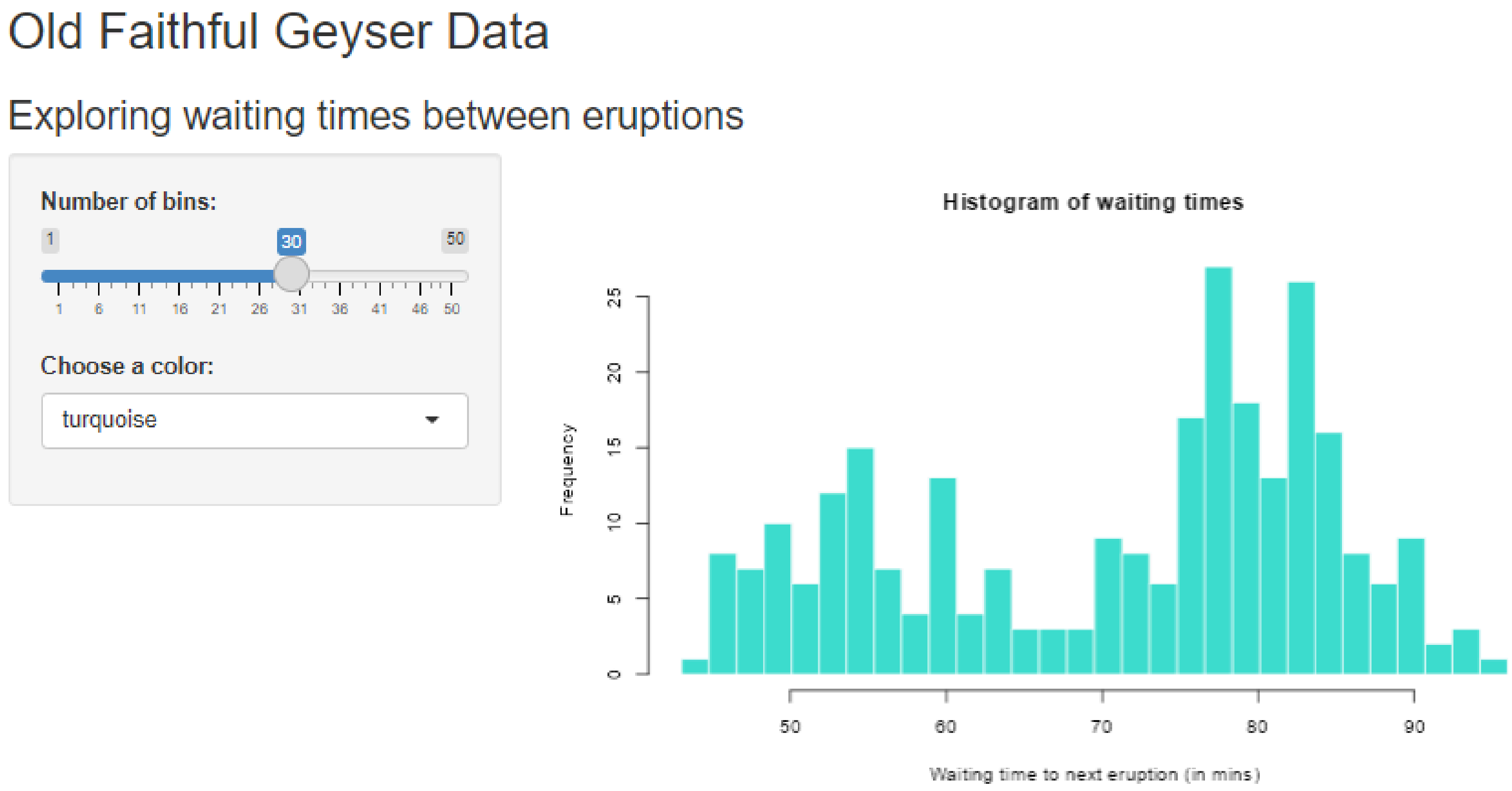
Server:

```r
col = input$color
```

✅ Solution: Exercise 2

- Let’s do this together.

See that if I don’t add the `input$color` in the server inside the `hist()` function, the color will not change.



Old Faithful Geyser Data

Exploring waiting times between eruptions

Number of bins: 30

Choose a color: turquoise

1
30
50
1
6
11
16
21
26
31
36
41
46
50
Histogram of waiting times
Frequency
0
5
10
15
20
25
50
60
70
80
90
Waiting time to next eruption (in mins)

Exercise 3: Add a Plot Type Selector ✍️

Ok! now let’s make this more challenging! Let’s give the user control over the **type of plot** they see!

📝 Your task:

- Add a `radioButtons()` input to let the user choose between:
  - `"Histogram"` of waiting times
  - `"Density` of eruption duration vs. waiting time
- Modify `renderPlot()` in `server.R` to change behavior based on selection

🔍 Hint:

UI:

```r
radioButtons("plot_type", "Choose a plot type:",
             choices = c("Histogram", "Density"))
```

In `server.R`, check the value of `input$plot_type` to decide which plot to draw.

✅ Solution: Exercise 3

Full server logic:

server.R

```r
function(input, output, session) {

    output$distPlot <- renderPlot({

        # generate bins based on input$bins from ui.R
        x    <- faithful[, 2]

        if (input$plot_type == "Histogram") {

        bins <- seq(min(x), max(x), length.out = input$bins + 1)

        # draw the histogram with the specified number of bins
        hist(x, breaks = bins, col = input$color, border = 'white',
             xlab = 'Waiting time to next eruption (in mins)',
             main = 'Histogram of waiting times')
        } else if (input$plot_type == "Density")
          {
          ggplot(faithful, aes(x=x)) +
            geom_density(alpha = 0.5, color = input$color) +
            labs(x = 'Waiting time to next eruption (in mins)',
                 title = 'Density Plot of Waiting Times') +
            theme_minimal()
          }

    })

}
```

Exercise 4: Adding an Intro Tab ✍️

It's always good practice to **explain what your app does**. For this, we can create an *intro tab* — like a README page — that gives your users helpful context.

1. Add a `tabsetPanel()` inside the `mainPanel()`.
2. Create two tabs:
    - One for the plot
    - One for the **Introduction**
3. In the Intro tab, write a short description of what the app does (in plain text or with HTML).

✅ Solution: Exercise 4

Here’s how your `ui.R` could look after adding the tabs:

```r
# Define UI for application that draws a histogram
navbarPage("Faithful Geyser Data - Customized",
           tabPanel("Introduction",
                      h3("Exploring the Faithful Geyser Data"),
                      p("This application allows you to visualize the waiting times between eruptions of the
                          You can choose between a histogram and a density plot, adjust the number of bins,

                    ),

           tabPanel("Plots",

                    # Sidebar with a slider input for number of bins
                    sidebarLayout(
                      sidebarPanel(
                        sliderInput("bins",
                                    "Number of bins:",
                                    min = 1,
                                    max = 50,
                                    value = 30),
                        selectInput("color", "Choose a color:", choices = c("turquoise", "plum", "orchid")),
                        radioButtons("plot_type", "Choose a plot type:",
                                     choices = c("Histogram", "Density"))
                      ),

                      # Show a plot of the generated distribution
                      mainPanel(
                        plotOutput("distPlot")
```

Exercise 5: Add a Theme ✍️

Want your app to look more polished? Shiny supports easy theming with the `{bslib}` package.

👉 **Your task**: Add a custom theme to your app!

🔧 Steps

1. **Load** the `bslib` package in your `global.R` file:

   ```r
   library(bslib)
   ```

   If you don't have it installed, run:

   ```r
   install.packages("bslib")
   ```

2. **Wrap your `navbarPage()`** in a `thematic` Bootstrap theme `ui.R` file:

   ```r
     theme = bs_theme(bootswatch = "minty") # Try "minty", "flatly", "journal", etc.
   ```

3. **Save** and re-run your app!

Add a Theme ✍️

✨ More bootswatch themes

- `"flatly"` (clean + modern)
- `"darkly"` (dark mode)
- `"minty"` (playful + bright)
- `"journal"` (serif style)
- Full list: https://bootswatch.com

Extra if there is time: Add Download Functionality ✍️

Let's allow users to **download the dataset** they are exploring!

📝 Your task:

- Add a **`downloadButton()`** to the UI so users can download the data.
- In `server.R`, define a **`downloadHandler()`** to write the `faithful` dataset as a CSV file.

🔍 Hint:

- You will use `downloadButton()` in the ui and `downloadHandler()` in the server.

✅ Solution: Exercise Extra

ui.R

```r
downloadButton("download_data", "Download Data")
```

server.R

```r
output$download_data <- downloadHandler(
  filename = function() { "faithful_data.csv" },
  content = function(file) {
    write.csv(faithful, file, row.names = FALSE)
  }
)
```

- The multiple file app with all the exercises we did is available here

Exercise Extra 2: Add Table Tab✍️

Let’s add a new tab to display the `faithful` dataset as a table.

📝 Your task:

- Create a new tab in `ui.R` called “Table”.
- In the new tab, use `tableOutput("data_table")` to display the dataset.
- In `server.R`, create a new output called `data_table` that renders the `faithful` dataset as a table.

🔍 Hint:

- Use `renderTable()` in the server to display the dataset.

✅ Solution: Exercise Extra 2

ui.R

```r
tabPanel("Table",
             h2("Faithful Geyser Data Table"),
             p("Below is the table of the Old Faithful geyser data. You can view the waiting times between e
             tableOutput("data_table"))
```

server.R

```r
output$data_table <- renderTable({
  faithful
})
```

- The multiple file app with all the exercises we did is available here
- Note: this is not super aesthetic, but you can use packages like `{DT}` or `{reactable}` to make it look better.

Share your Shiny app

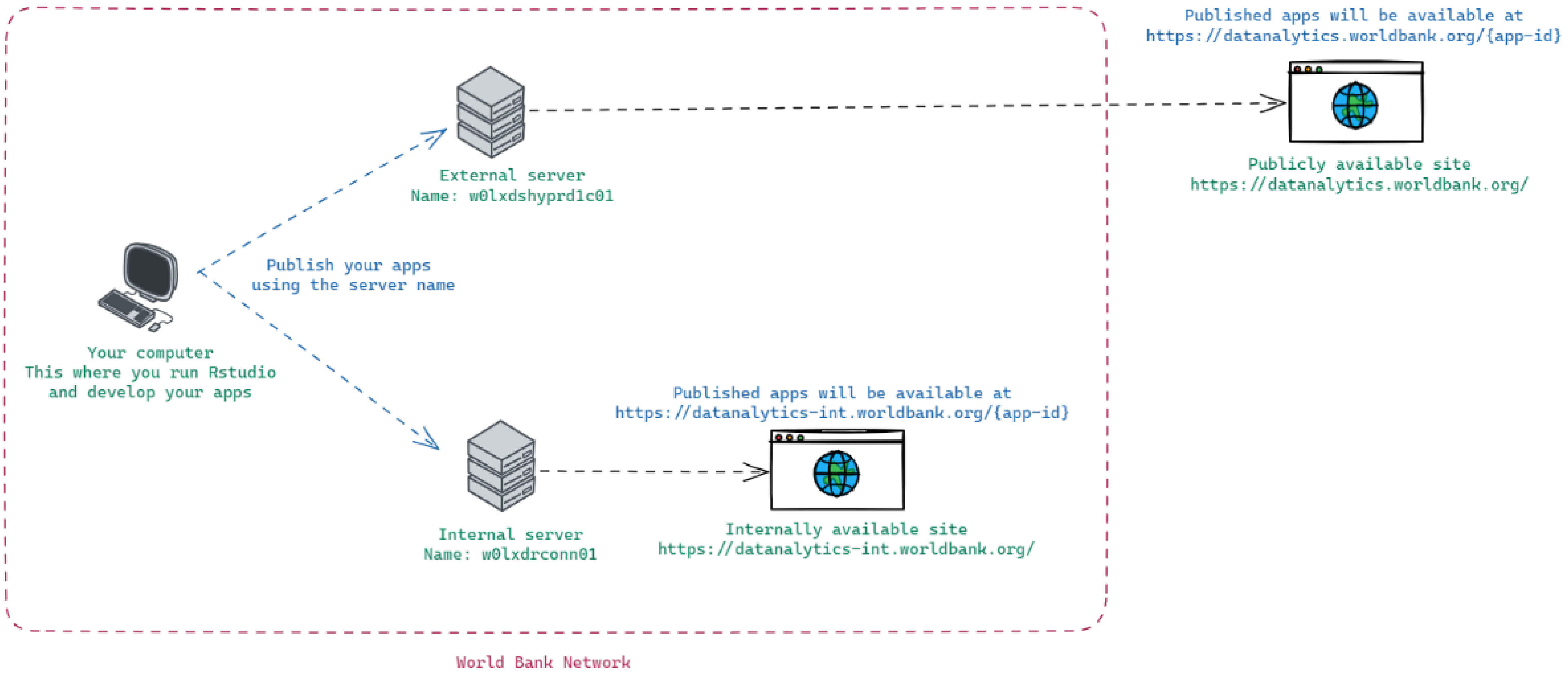
- 🏢 **Posit Connect Internal Server**
    - Recommended option for **secure deployment within the Bank**
    - Content is deployed on Bank server behind a firewall, **only accessible to Bank employees**
- 🌐 **Posit Connect Public Server**
    - Content is deployed on Bank server behind a firewall, but accessible to all.
    - **Only display public use data**

For both:

- Push-button publishing from RStudio or publish directly from GitHub
- Request Posit Connect access as a Software Request
- Learn more, Internal Resources

Share your Shiny app

6 World Bank PositConnect architecture



Published apps will be available at
https://datanalytics.worldbank.org/{app-id}

External server
Name: w0lxdshyprd1c01

Publicly available site
https://datanalytics.worldbank.org/

Publish your apps
using the server name

Your computer
This where you run Rstudio
and develop your apps

Published apps will be available at
https://datanalytics-int.worldbank.org/{app-id}

Internal server
Name: w0lxdrconn01

Internally available site
https://datanalytics-int.worldbank.org/

World Bank Network

Thank you! 🙏

Additional Resources 📚

Want to go further with Shiny? Here are some helpful resources:

- 🧪 **Shiny Tutorial** (Official Getting Started Guide) here
- 📘 **Mastering Shiny** by Hadley Wickham (Free online book) here
- 💡 **Shiny Widgets Gallery** here
- 🧩 **Awesome Shiny Extensions** (Community plugins) here
- 🌐 **Shiny Community** (Forums, discussions) here
- 📦 **Building Web Applications** (Training) here
- 🧱 **Adding multiple objects in layout** here

Some examples ✨

- Shiny App Gallery here
- California Schools Climate Hazards Dashboard here
- New Zealand Trade Intelligence Dashboard here
- Locating Blood Banks in India here
- Understanding voters’ profile in Brazilian elections here