Introduction to Shiny

NYC Data Science Academy

Outline

- **A Quick Tour of Shiny**
- Building a Shiny App from Scratch
  - Create an Empty Shiny App
  - Build the Basic UI
  - `*input` and `*output` functions
  - `render*` functions
- Improving your Shiny app
  - Use global.R for preprocessing
  - Reactive Expression
  - Update Input widgets
  - Reactive Observer

What is Shiny

- R package from RStudio
- Web application framework for R
- R code → interactive web page
- No HTML/CSS/JavaScript knowledge required (but better to know some)

Before we begin, you'll need to have the shiny package installed.

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

Shiny User Showcase

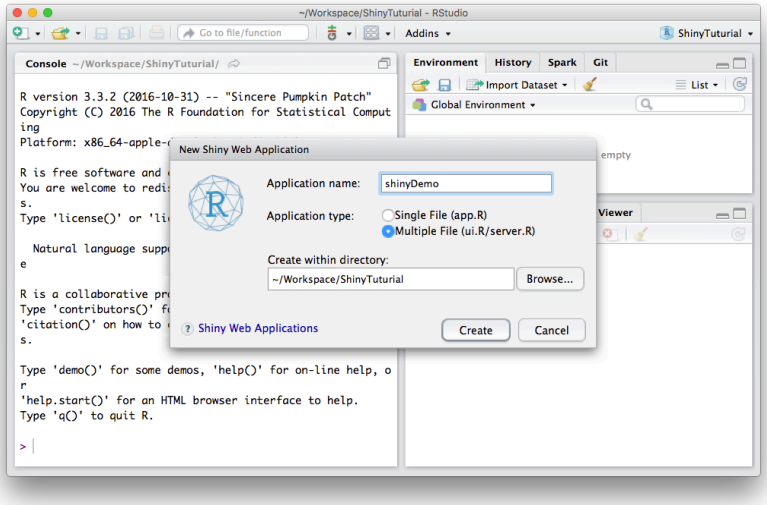
A Quick Tour of Shiny

You can create a new Shiny app using RStudio's menu by selecting:

- *File* > *New File* > *Shiny Web App*....

If you do this, RStudio will let you choose if you want a single-file app (`app.R`) or a two-file app (`ui.R` + `server.R`).

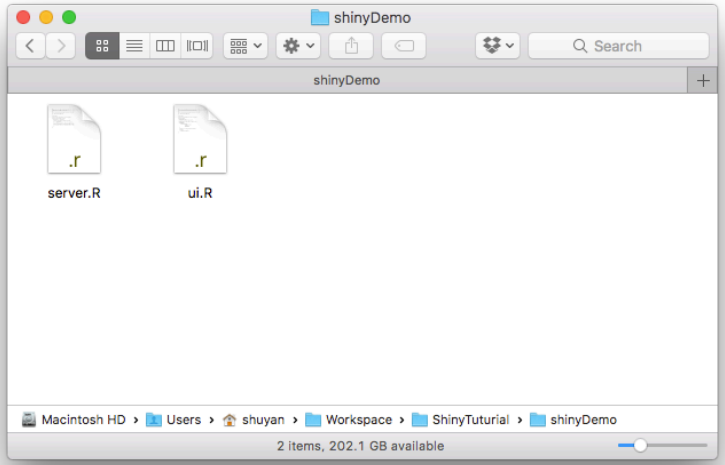
Now let's give your app a name called `shinyDemo` and then hit `Create`:



A Quick Tour of Shiny

If you choose a two-file app then RStudio will create two files `ui.R` and `server.R` saved in the same directory.

The name of of directory is the name of your shiny app, in our case it is `shinyDemo`.



A Quick Tour of Shiny

RStudio will initialize a simple functional Shiny app with the following code in the two scripts

ui.R

```
library(shiny)

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

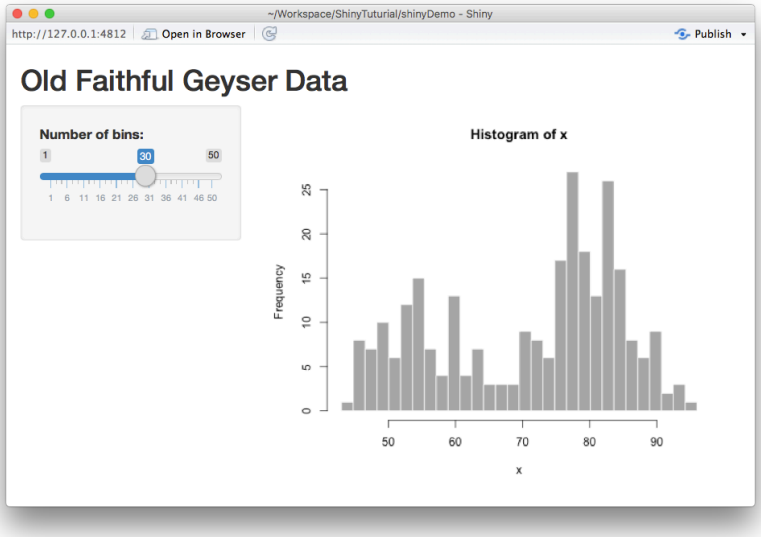
A Quick Tour of Shiny

server.R

```
library(shiny)

shinyServer(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')
  })

})
```

A Quick Tour of Shiny

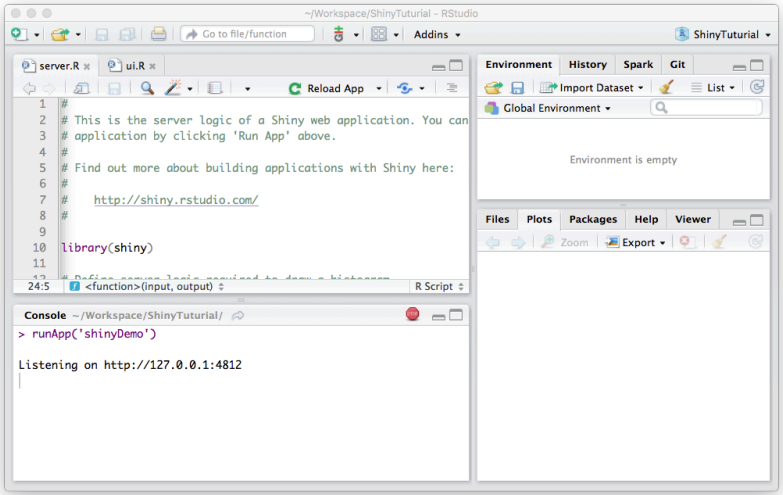
Click the `Run App` button or execute `runApp('shinyDemo')` in the Console and your shiny app should run in a Browser:



A Quick Tour of Shiny

While your Shiny is running, your R session is busy with powering the app, and won't response to any commands in the console.

To stop the Shiny app, you can either close the browser that has Shiny app running on it, or click the red stop sign appeared at the top right of the console.



Outline

- A Quick Tour of Shiny
- **Building a Shiny App from Scratch**
  - Create an Empty Shiny App
  - Build the Basic UI
  - `*input` and `*output` functions
  - `render*` functions
- Improving your Shiny app
  - Use global.R for preprocessing
  - Reactive Expression
  - Update Input widgets
  - Reactive Observer

Create an Empty Shiny App

Now let's build a Shiny app from scratch.

Instead of creating an Shiny app using RStudio's menu, we can also get started by the following steps:

1. create a new empty directory (project directory), name it `flights`,
2. under the top level of your project directory, create two empty R scripts and name them as `ui.R` and `server.R`.

```
flights\
  |-- ui.R
  |-- server.R
  |
  ...
```

Create an Empty Shiny App

The following template provides a working minimal Shiny app that initialize an empty UI and an empty server. Copy them into your `ui.R` and `server.R`.

ui.R

```
library(shiny)

fluidPage(

)
```

server.R

```
library(shiny)

function(input, output) {

}
```

Create an Empty Shiny App

After saving the files, RStudio should recognize that this is a Shiny app, and you should see the usual Run button at the top change to `Run App`.

Build the Basic UI

Let's start populating our app with some elements visually. This is usually the first thing you do when writing a Shiny app - add elements to the UI.

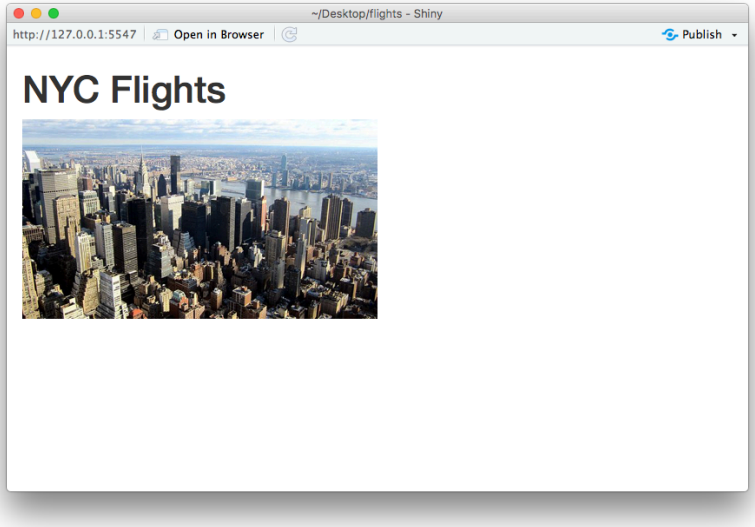
- Add plain text to the `fluidPage()`

```
library(shiny)

fluidPage(
  "NYC Flights"
)
```

Build the Basic UI

Shiny has many functions for constructing HTML documents. Now try replacing the `fluidPage()` function in your UI to show the title and an image with:

```
fluidPage(
  h1("NYC Flights"),
  img(src="https://upload.wikimedia.org/wikipedia/commons/thumb/3/38/Bli
      width="50%")
)
```



Build the Basic UI

- Use HTML elements in UI

Those html functions all work in a similary way by providing wrappers to HTML tags with the equivalent name.

```
print(h1("NYC Flights"))
```

```
## <h1>NYC Flights</h1>
```

You can also pass arguments to those wrapper functions:

```
print(img(src="https://upload.wikimedia.org/wikipedia/commons/thumb/3/38
          width="50%"))
```

```
## <img src="https://upload.wikimedia.org/wikipedia/commons/thumb/3/38/Blick_auf
```

Build the Basic UI

In fact you can add any arbitrary HTML tag using the tags object. For example, the following html embeds an youtube video to the web:

```
<iframe width="640" height="360" src="https://www.youtube.com/embed/pXU_GY7hjPc"
```

In shiny you can do:

```
fluidPage(h1("NYC Flights"),
          tags$iframe(src="https://www.youtube.com/embed/pXU_GY7hjPc",
                      width="640", height="360"))
```

Build the Basic UI

If you would like to include local images and/or any .css/.js files, you need to place them under the subdirectory named `www`.

For example, to include an image named `flights.jpg`, your directory structure should look like:

```
  flights\
    |-- www\
    |   |-- flights.jpg
    |-- ui.R
    |-- server.R
    |
    ...
```

and you then can include the image in your app using:

```
img(src="flights.jpg", ...)
```

Build the Basic UI

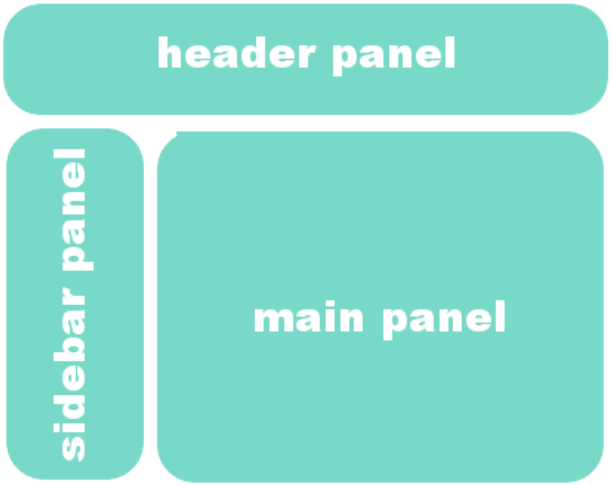
For a complete list of available tags and the use cases, please go to Shiny HTML Tags Glossary.

Build the Basic UI

So far, by just adding text and HTML tags, everything is unstructured and the elements simply stack up one below the other in one column.

A webpage layout is very important to give better look to your website. Here we'll use `sidebarLayout()` to add a simple structure.

```
fluidPage(
  titlePanel(...),
  sidebarLayout(
    sidebarPanel(...),
    mainPanel(...)
  )
)
```



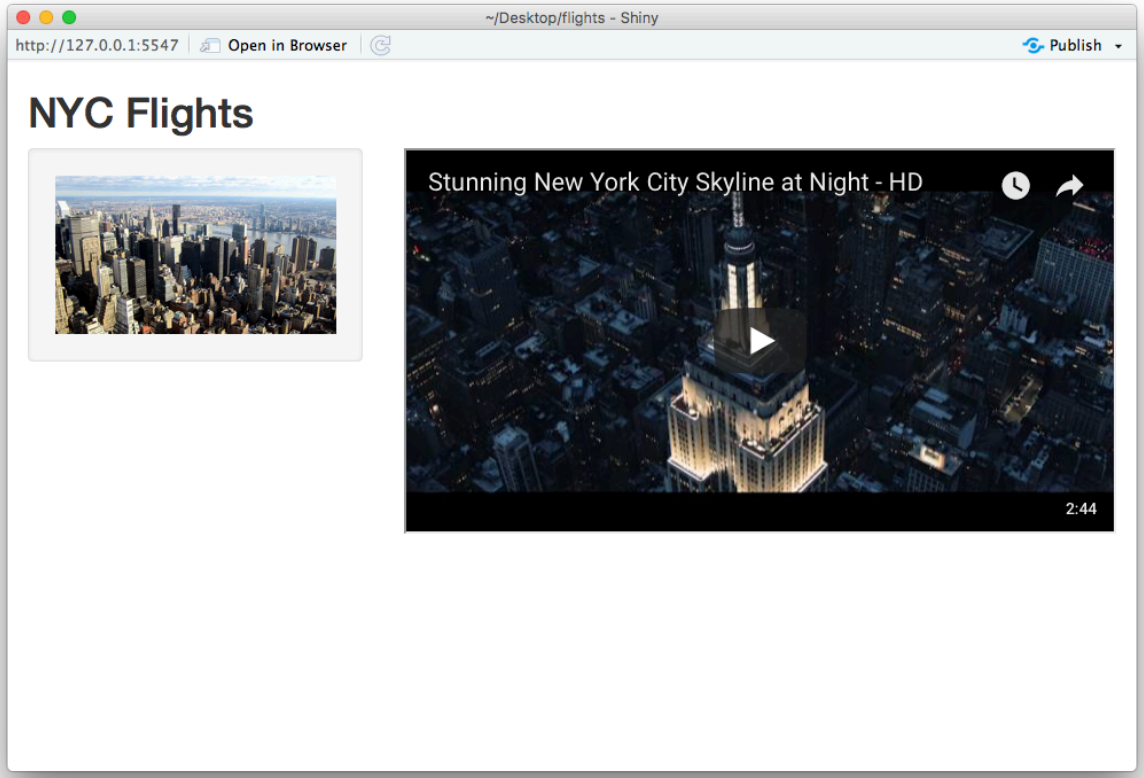
Build the Basic UI

Now change your `ui.R` as follows:

```
fluidPage(
  titlePanel("NYC Flights"),
  sidebarLayout(
    sidebarPanel(
      img(src="https://upload.wikimedia.org/wikipedia/commons/thumb/3/38
          width="100%")
    ),
    mainPanel(
      tags$iframe(src="https://www.youtube.com/embed/pXU_GY7hjPc",
                  width="100%", height="280px")
    )
  )
)
```

Build the Basic UI

You shiny app should now look like:



Build the Basic UI

- With the Sidebar Layout, you cannot do much customization with your UI layout.
- There are many other UI funcitons in Shiny that gives you much more flexibility in UI design.

For detailed information about shiny layouts, please refer to Application layout guide.

`*input` and `*output` in UI

- `*Input()` function, also called control widgets, gives users a way to interact with a Shiny app.

  For examples, please go to Shiny Widgets Gallery.

- `*Output()` function add an R object to the user-interface.

*input and *output Syntax

- `selectizeInput(inputId = "origin", label = "Departure airport", ...)`
  - `selectizeInput`: type of input widget
  - `inputId`: input Id, for internal use only, must be unique!
  - `label`: label to display, value can be NULL
  - `... (the rest)`: input specific arguments
- `plotOutput(outputId = "count", ...)`
  - `plotOutput`: the type of output to display
  - `outputId`: the Id of the output object
  - `the rest`: output specific arguments

*input and *output in UI

If we want to use any external dataset for our shiny apps we need to put the data file under the same project directory and read the data via the shiny scripts.

We will use `flights14.csv` in this example, the directory should look like:

```
flights\
  |-- ui.R
  |-- server.R
  |-- flights14.csv
  |
  ...
```

Note: you can also put you data files under a sub-folder named `data` to make your project more organized.

After we include the `csv` file in the project directory, the shiny app will have access to the data via:

```
flights <- read.csv(file = "./flights14.csv")
```

*input and *output in UI

Now replace the `img(...)` with two `selectizeInput()`s inside the `sidebarPanel` and replace the `tags$iframe(...)` with a `plotOutput()` inside the `mainPanel`:

```
flights <- read.csv(file = "./flights14.csv")
fluidPage(
  titlePanel("NYC Flights 2014"),
  sidebarLayout(
    sidebarPanel(
      selectizeInput(inputId = "origin",
                     label = "Departure airport",
                     choices = unique(flights$origin)),
      selectizeInput(inputId = "dest",
                     label = "Arrival airport",
                     choices = unique(flights$dest))
    ),
    mainPanel(plotOutput("count"))
  )
)
```

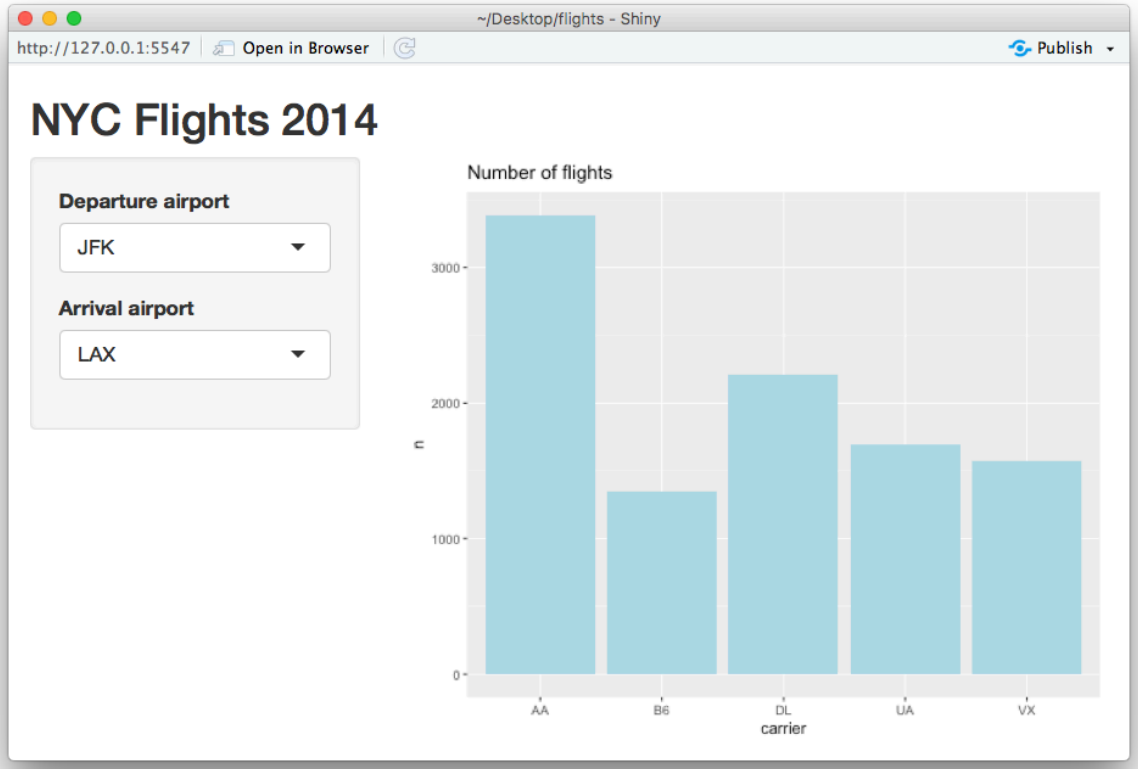
Add `render*()` to the Server

We then need to tell the server how to assemble inputs into outputs. In `server.R`, modify the `function(input, output)` with the code below:

```
library(shiny)
library(dplyr)
library(ggplot2)

flights <- read.csv(file = "./flights14.csv")

function(input, output) {
  output$count <- renderPlot(
    flights %>%
      filter(origin == input$origin & dest == input$dest) %>%
      group_by(carrier) %>%
      count() %>%
      ggplot(aes(x = carrier, y = n)) +
      geom_col(fill = "lightblue") +
      ggtitle("Number of flights")
  )
}
```

Add `render*()` to the Server

Run the app again and you should get the following output:



How to Write the Server Function

- Save objects you want to display to `output$outputId`:

```
function(input, output) {
  output$count <- ## code
}
```

The `outputId` in `output$outputId` should match the one in the `*Output("outputId")` in `ui.R`.

Here the `outputId` is `"count"`.

How to Write the Server Function

- Build (reactive) objects to display with `render*()`:

```
function(input, output) {
  output$count <- renderPlot(
    flights %>%
      filter(origin == "JFK" & dest == "LAX") %>%
      group_by(carrier) %>%
      count() %>%
      ggplot(aes(x = carrier, y = n)) +
      geom_col(fill = "lightblue") +
      ggtitle("Number of flights")
  )
}
```

The `renderPlot()` function will send a plot object to `plotOutput()` through `output$count`.

How to Write the Server Function

- Access input values with `input$inputId`:

In stead of using `"JFK"` and `"LAX"` for origin and dest, we can access users' inputs by changing:

```
function(input, output) {
  output$count <- renderPlot(
    flights %>%
      filter(origin == input$origin & dest == input$dest) %>%
      group_by(carrier) %>%
      count() %>%
      ggplot(aes(x = carrier, y = n)) +
      geom_col(fill = "lightblue") +
      ggtitle("Number of flights")
  )
}
```

The `inputId` in `input$inputId` should match the one in the `*Input(inputId, ...)` in `ui.R`. Here the `inputId`s are `"origin"` and `"dest"`.

Aside: one file vs. two file

app.R

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

server <- function(input,output){
  output$count <- renderPlot({
    ...
  })
}

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

ui.R

```
fluidPage(
  titlePanel(...),
  sidebarLayout(
    sidebarPanel(...),
    mainPanel(...)
))
```

server.R

```
function(input, output) {
  output$count<-renderPlot({
    ...
  })
}
```

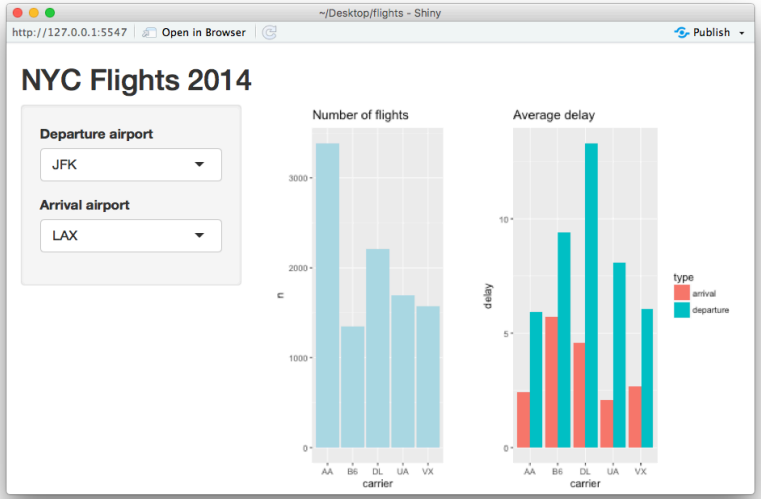
Exercise 1

1. Add another ggplot2 barchart to show the average delays between airports. You can change your UI layout to make two graphs side by side using (the sum of widths in a fluidRow should be equal or less than 12):

```
fluidPage(
  titlePanel("NYC Flights 2014"),
  sidebarLayout(
    ...
    ,
    mainPanel(
      fluidRow(
        column(5, plotOutput("count")),
        column(7, plotOutput("delay"))
      )
    )
  )
)
```

Exercise 1

Your app now should look similar to:



For all the functions in Shiny, please go to Function reference.

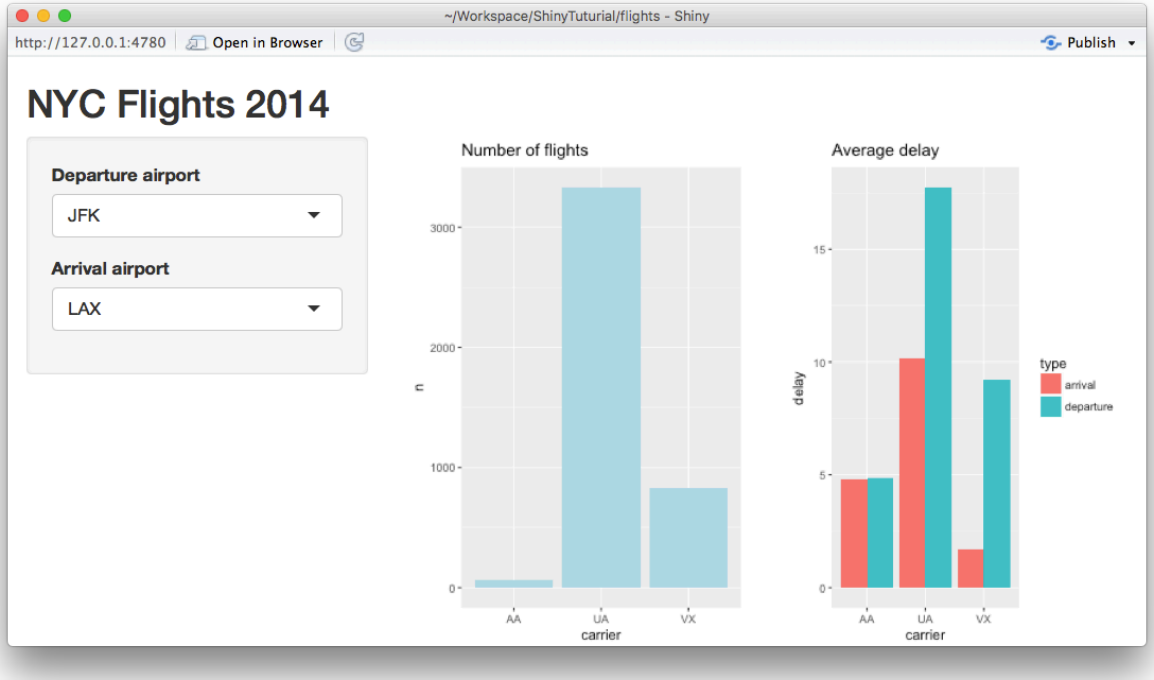
Outline

- A Quick Tour of Shiny
- Building a Shiny App from Scratch
  - Create an Empty Shiny App
  - Build the Basic UI
  - `*input` and `*output` functions
  - `render*` functions
- **Improving your Shiny app**
  - Use global.R for preprocessing
  - Reactive Expression
  - Update Input widgets
  - Reactive Observer

NYC Flights 2014

In this example we will be using the NYC Flights 2014 dataset.

The flights app contains two input widgets and two outputs. It works but still needs improvement.



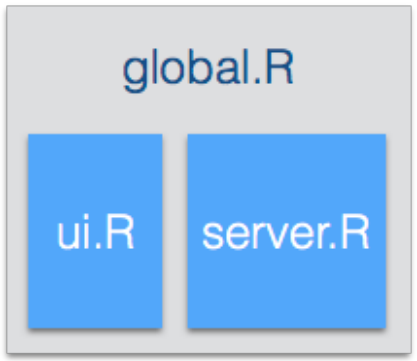
Use global.R for Preprocessing

- Issue: Data is required in both `ui.R` and `server.R`. We loaded the data file twice, one in `ui.R` and one in `server.R`, since they use different environments.
- Goal: Load global variables such as data only once so both `ui.R` and `server.R` can access to it.
- Solution: Use `global.R` to load and preprocess data.

Use global.R for Preprocessing

`global.R` is an optional script in shiny which always runs before `ui.R` and `server.R` if provided.

Objects defined in `global.R` are loaded into the global environment of the R session and are accessible to both `ui.R` and `server.R`.



Use global.R for Preprocessing

Now create a new file `global.R` and save the following chunk of code in it:

```
library(data.table)
flights <- fread(file = "./flights14.csv")
```

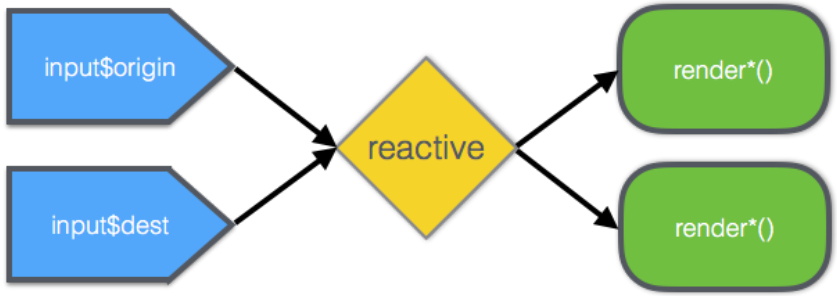
Then delete those lines from both `ui.R` and `server.R` and test your app.

Reactive Expression

- Issue: The following chunk of code are calculated in both `renderPlot()` function.

```
flights %>%
filter(origin == input$origin & dest == input$dest) %>%
group_by(carrier) %>%
summarise(...)
```

- Goal: Calculate it once and share it with the two `renderPlot()` functions.
- Solution: Use reactive expressions.



Reactive Expression

A reactive expression is an R expression that uses widget inputs (and/or other reactive expressions) and returns an R object. The reactive expression will update this object whenever the original widget changes.

To create a reactive expression use the `reactive` function, for example:

```
function(input, output) {

  flights_delay <- reactive({
    flights %>%
      filter(origin == input$origin & dest == input$dest) %>%
      group_by(carrier) %>%
      summarise(n = n(),
                departure = mean(dep_delay),
                arrival = mean(arr_delay))
  })

  ...
}
```

Reactive Expression

The reactive expression works like a function: you call it inside a `render*` function or another reactive expression to get its value.

With the reactive expression that we defined, our `renderPlot` functions can now be rewritten as:

```
function(input, output) {
  flights_delay <- reactive(...)
  output$delay <- renderPlot(
    flights_delay() %>%
      gather(key = type, value = delay, departure, arrival) %>%
      ggplot(aes(x = carrier, y = delay, fill = type)) +
      geom_col(position = "dodge") + ggtitle("Average delay")
  )
  output$count <- renderPlot(
    flights_delay() %>%
      ggplot(aes(x = carrier, y = n)) +
      geom_col(fill = "lightblue") + ggtitle("Number of flights")
  )
}
```

Update Input widgets

- Issue: Given a departure airport, not all of the arrival airports are available, for example, there's no flight between LGA and LAX.
- Goal: Update the arrival airport list according to the selected departure airport.
- Solution: `updateSelectizeInput()` for each `session`.

Update Input widgets

Shiny server functions can optionally include session as a parameter (e.g. function(input, output, session)).

The session object is an environment that can be used to access information and functionality relating to the session (a client based environment).

```
function(input, output, session) {
  updateSelectizeInput(
    session, "dest",
    choices = unique(flights[origin == "LGA", dest]),
    selected = unique(flights[origin == "LGA", dest][1]))
  ...
}
```

What happens if you change LGA to input$origin?

Note: The `update*()` function requires session as the first parameter.

observers

- Issue: Received an Error: Operation not allowed without an active reactive context.
- Reason: Shiny requires all the `input$*` to be wrapped inside a reactive exression.
- Solution: use a reactive observer.

Difference between observers and reactive expressions

- *Reactive expression* yields a result, which can be used as an input to other reactive expressions.
- *Observer* has no result created and is only useful for its side effect.

Execution strategy:

- *Reactive expressions* use **lazy evaluation**: when their dependencies change, they don't re-execute right away but rather wait until they are called by someone else.
- *Observers* use **eager evaluation**: as soon as their dependencies change, they schedule themselves to re-execute.

observe() function

Now let's wrap the code that gives an error with the `observe()` function:

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

  observe({
    dest <- unique(flights[origin == input$origin, dest])
    updateSelectizeInput(
      session, "dest",
      choices = dest,
      selected = dest[1])
  })

  ...
}
```

Exercise 2

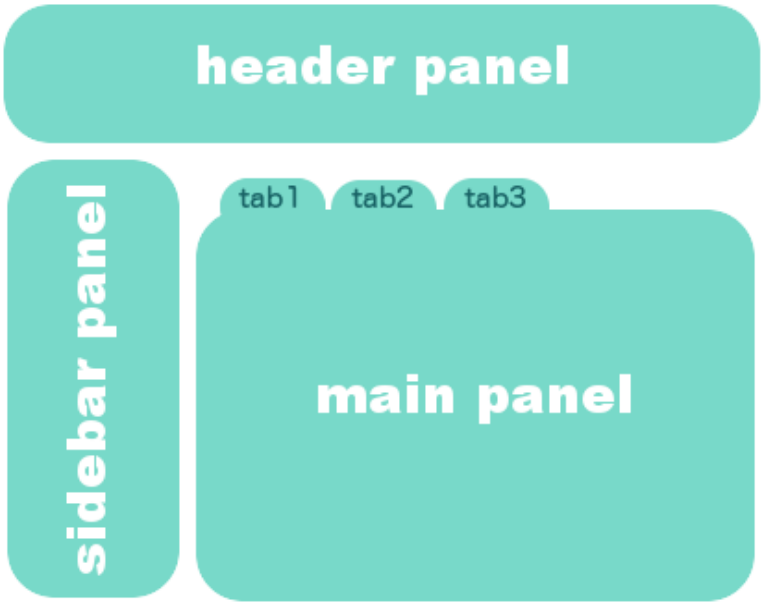
1. Add a input widget to allow user select different month. Then modify the server code accordingly.

2. Add another tab and display the dataframe you used for the two plots.

    - You can use `renderTable` and `tableOutput` to display the dataframe.
    - The next slide shows how to have multiple tabs in the main panel.

Exercise 2

```
fluidPage(
  titlePanel(...),
  sidebarLayout(
    sidebarPanel(...),
    mainPanel(
      tabsetPanel(
        tabPanel("tab1", ...),
        tabPanel("tab2", ...),
        tabPanel("tab3", ...)
      )
    )
  )
)
```



For other layouts please refer to Application layout guide.