IN THIS CHAPTER

- Introducing shiny
- Looking at a simple `shiny` project
- Developing your project
- Coming up with a more complex project

Chapter 1
Working with a Browser

As I emphasize throughout this entire book, R is rich with opportunities for visualizing data. In this chapter, I show how to create R applications whose visualizations depend on user input. I also show how to present these applications in a browser so that web users can interact with them. Putting an R application in a browser is a helpful way to share data and analyses. And you don't have to know HTML or JavaScript to get the job done!

Getting Your Shine On

A creation of RStudio honchos, `shiny` is the package that enables interactive, browser-based R applications. Use RStudio to install it in the usual way. On the Packages tab, click Install and then type **shiny** into the Install Packages dialog box. After the package finishes downloading, select the check box next to `shiny` on the Packages tab, or type

```
> library(shiny)
```

A couple of words about architecture before I move on and show how to create your first `shiny` project. Behind any web page with a `shiny` app is a computer that serves that page. The computer is running the R code (also known as a *script*) that creates the page. Though the computer can be a server that operates via the cloud, for the apps I show you in this chapter, the server is your laptop.

Creating Your First shiny Project

A `shiny` application is a directory that contains a file with R code. So, in your working directory, create a new directory called `shinydir1`.

RStudio gives you an easy way to do this: With the `shiny` package installed, choose File➪New File➪➪Shiny Web App.

This menu command opens the New Shiny Web Application dialog box, shown in Figure 1-1.

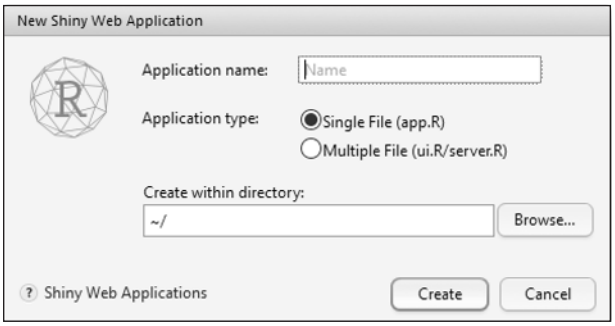

FIGURE 1-1: The New Shiny Web Application dialog box.

In the Application Name box, I enter a descriptive name for the app I'm about to create. I'm creating an interactive histogram that shows random sampling from a uniform distribution, so I type **UniformRandom** (no spaces!).

For the Application Type option, I leave the Single File (app.R) radio button selected.

Finally, I create the directory. I click the Browse button to open the Choose Directory dialog box, which you see in Figure 1-2.

In this dialog box, I create a new folder called `shinydir1` and click Select Folder. This closes the Choose Directory dialog box. Back in the New Shiny Web Application dialog box, I click Create.

After you complete these steps, you'll notice that the tab in the Scripts pane is now titled `app.R`. Every `shiny` app tab is labeled `app.R`. (Different `app.R` applications reside in different directories.) You'll also notice that an R script for a sample `shiny` app appears in the pane. Figure 1-3 shows you what I mean.

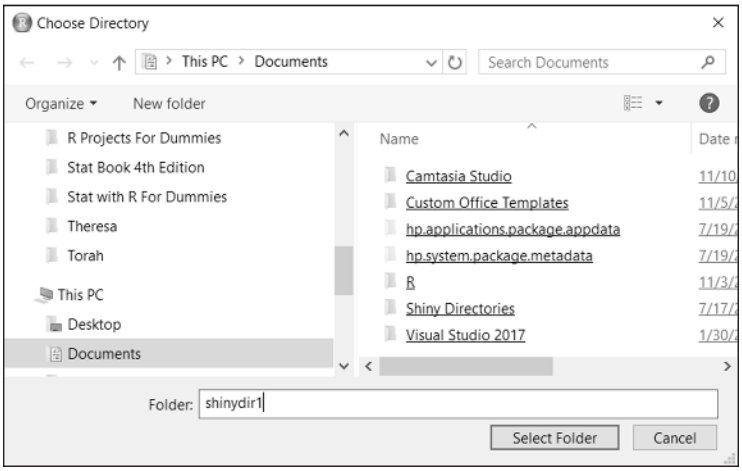

FIGURE 1-2: The Choose Directory dialog box.

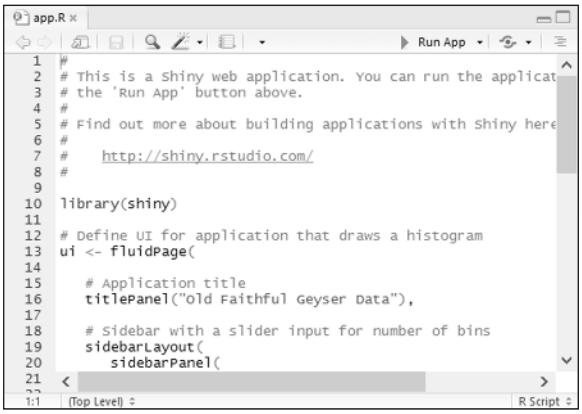

FIGURE 1-3: The Scripts pane after clicking Create in the New Shiny Web Application dialog box.

As the comment lines in the Scripts pane tell you, you can run this sample app by clicking the Run App button at the top of the pane. I'll leave it to you to run this application and see how the application reflects the code.

In this section, however, I delete the sample code and develop a similar (but somewhat more elaborate) application that teaches you some additional R skills as I explore `shiny`'s capabilities. With the Scripts pane active, I press Ctrl+A and press Delete. Now I have an empty Scripts pane.

The code for a `shiny` app has two main components: a *user interface* and a *server.*

The first order of business is to create a function that defines the *user interface* — the page that the user sees and interacts with. The fundamental structure of this script is

```
ui <- type_of_page()
```

Several types of pages are possible. Arguments in the parentheses determine the appearance and functionality of the page.

Then you create a set of instructions for the server to execute when the user interacts with the user interface. One way to begin is

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

Inside the curly brackets, put the instructions you create.

Finally, the function

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

ties together the `ui` and the `server` into a `shiny` application.


TECHNICAL STUFF

In the early days of `shiny`, it was necessary to create one file for the user interface and another for the server (including the `shinyApp()` function) and to store both in the directory. You can still do that (by choosing the Multiple File radio button in the New Shiny Web Application dialog box). Nowadays, only one file is necessary, and that's the way I do it in this chapter.

Figure 1-4 shows what your first `shiny` project looks like when all the pieces are in place.

It's a simple app, and it's typical of first projects with this package. The user manipulates a slider to determine the number of values to sample in a uniform distribution. The minimum value of the distribution is 0, and the maximum value is 1.

The histogram shows the results of the sampling. The minimum number of values is 25, the maximum number is 1,000, and the default is 500. Figure 1-4 shows the app in a window that RStudio opens.

To see the app in a browser, click Open in Browser in the upper left corner. (Spoiler alert: It looks pretty much the same.) As we proceed, I show you `shiny` apps in RStudio windows because they look better in the confines of the pages you're reading. Just bear in mind that it's easy enough to see the browser version by clicking Open in Browser.

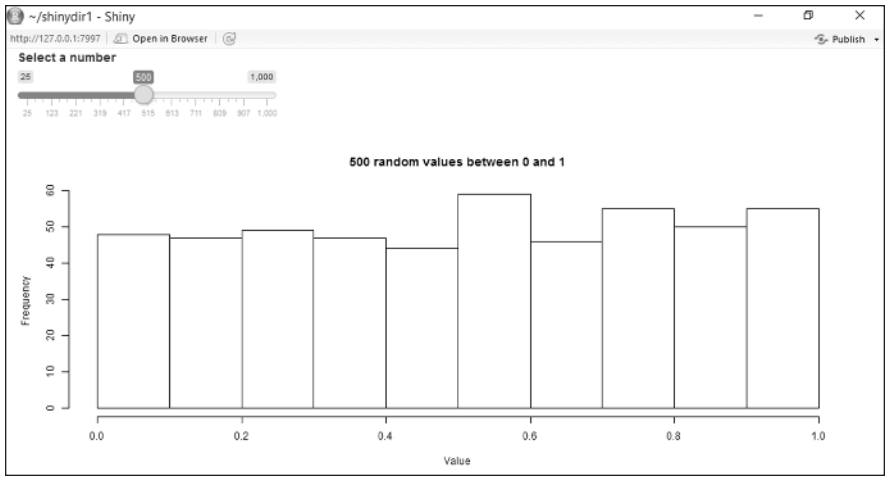

FIGURE 1-4: Your first `shiny` project.

The user interface

First things first. To define the user interface, I specify the type of page. For this application, I want a page that changes with the width of the browser: If I make the browser narrower, for example, I want the appearance of the page to change accordingly. This type of page is *fluid*, so the function that creates it is called `fluidPage()`:

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

And that's the beginning of the user interface.

Next, I need a function that defines the slider and the input (the result of moving the slider) and another function that sets up the output. I put those two functions inside the parentheses.

For the slider, it's

```
sliderInput(inputId = "number",
            label = "Select a number",
            value = 500, min = 25, max = 1000)
```

The first argument establishes an identifier for the number the user selects by moving the slider. In the upcoming `server()` function, I refer to it as `input$number`.

The second argument adds the instruction above the slider. The remaining arguments set the default number, the minimum number, and the maximum number of values to sample from the uniform distribution.

Finally, I reserve an area for the output:

```
plotOutput("hist")
```

At this point, the app doesn't know what kind of output to plot. All it knows is that `"hist"` is the name of the output.

The user interface code is

```
ui <- fluidPage(
  sliderInput(inputId = "number",
              label = "Select a number",
              value = 500, min = 25, max = 1000),
  plotOutput("hist")
)
```

Think of `sliderInput()` as an input function and `plotOutput()` as an output function.

What does this code actually do? In the Scripts pane, highlight `fluidPage()` and all its arguments (don't include `ui <-` in your highlighting). Then press Ctrl+Enter to run the highlighted code. The result? A lot of HTML in the Console pane. This shows you that the user interface code generates a web page.

The server

As I point out earlier in this chapter, the starting point for the `server` is

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

The first thing to put in the curly brackets is an R expression that represents the output. In the user interface, the name of the output is `"hist"`. Here in the output, I refer to it as `output$hist`.

That expression receives the value of a function called `renderPlot()`, which, unsurprisingly, renders the plot.

A word about `renderPlot()`. The syntax of this function is

```
renderPlot({})
```

Inside `renderPlot`'s curly brackets, you add as many lines of code as necessary to, well, render the plot. In this application, the base R graphics function `hist()` does the honors, as described in Chapter 1 of Book 2:

```
server <- function(input, output) {
  output$hist <- renderPlot({ hist(runif(input$number,min=0,max=1),xlab="Value",
main=paste(input$number,"random values between 0   and 1"))
   })
  }
```

That first argument to `hist()` is `runif()`. Do *not* pronounce it "run if"! It's not a *run* statement combined with an *if* statement or anything like that. Instead, think of the *r* as *random* and *unif* as *uniform.* This is R's way of saying, "Randomly sample from a uniform distribution." (The correct pronunciation is "r unif.") How would R say, "Randomly sample from a normal distribution"? If you guessed `rnorm()`, you're absolutely right.

The first argument to `hist()` indicates that the data for the histogram comes from a random sample of values from a uniform distribution. How many values are in the sample? `input$number`, that's how many. The next two arguments to `runif()` set the distribution's minimum value to 0 and its maximum value to 1.

Now for the remaining arguments for `hist()`. If you've completed the examples in Chapter 1 of Book 2, you'll remember that `xlab` labels the *x*-axis, and `main` provides a title. Within `main`, I use the `paste()` function to add the value of `input$number` to the beginning of the title. The result is that the histogram title (as well as the histogram) changes each time the user moves the slider to a new number.

Final steps

To tie the user interface to the server, I add

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

The entire script (including the `library()` function at the beginning) is

```
library(shiny)
ui <- fluidPage(
  sliderInput(inputId = "number",
              label = "Select a number",
              value = 500, min = 25, max = 1000),
  plotOutput("hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({      hist(runif(input$number,min=0,max=1),xlab="Va
               lue",main=paste(input$number,"random values between 0 and 1"))
   })
  }
shinyApp(ui = ui, server = server)
```

Save the code (press Ctrl+S or choose File➪Save) and then click the Run App button. That opens the display shown earlier, in Figure 1-4, and puts this in the Console pane:

```
> runApp('shinydir1/UniformRandom')

Listening on http://127.0.0.1:3328
```

The second line means that R is waiting for the user to do something. (Move the slider, in other words.) The URL on your machine will no doubt be different from the one on mine.

To end the session with the application, press Esc or close the RStudio window that shows the page. You can also click the little red stop sign in the upper right corner of the Console pane.

Getting reactive

I can write the server in a different way. Instead of this:

```
server <- function(input, output) {
  output$hist <- renderPlot({ hist(runif(input$number,min=0,max=1),xlab="Value",
main=paste(input$number,"random values between 0   and 1"))
   })
  }
```

I can write this:

```
server <- function(input, output) {
  histdata <- reactive({
    runif(input$number,min=0,max=1)
    })

  output$hist <- renderPlot({
    hist(histdata(),xlab="Value",
    main=paste(input$number,"random values between 0 and 1")
  )
  })
}
```

It accomplishes the same thing. Why bother setting a variable called `histdata` for the `runif()` function? And what's that `reactive({})` deal? And, finally, why do I have parentheses after `histdata` in the first argument to the `hist()` function?

Creating the `histdata` variable enables me to use the results of sampling from the uniform distribution for additional outputs — not just for the histogram. For example, I might want to add the data's mean, median, and standard deviation to the `shiny` app. I show you how to do that in just a moment.

What about `reactive({})`? To make the `shiny` app responsive to user input, I have to create `histdata` in a *reactive context* so that the variable can *react* to the input (when the user moves the slider to change the value of `input$number`, in other words). Accordingly, `reactive({})` provides that context.

"But wait a second," you might exclaim. "In the original version, I was able to explicitly use `runif()` in `renderPlot({})`. Why is that?" Because `renderPlot({})` is a reactive context. (The curly brackets are a giveaway.) For that reason, changes in `input$number` show up as changes in the histogram plot. If `renderPlot({})` is the only reactive context I use, it's not necessary to have another reactive context and create `histdata`.

TIP

In `shiny`, every `render` function is a reactive context.

Now for the parentheses next to `histdata` in `hist()`. When I create a reactive variable like `histdata`, I create an object I can *call* to see whether changes have occurred (in this case, to `input$number`), and it returns the changes. If it's callable and it returns something, it's a function, and to indicate that, I add the parentheses. So `histdata` is the reactive variable I define here, and when I use it again, it's `histdata()`. Got it?

TIP

I can't emphasize this enough: When you create a variable in a reactive context, *you must add the parentheses whenever you use it*. Forgetting to do that is the biggest roadblock when you're starting out with `shiny`.

Now my objective is to create a `shiny` app that shows not just the histogram of the sample from the uniform distribution but also the sample mean, median, and standard deviation. The app will look like Figure 1-5. The mean, median, and standard deviation are below the histogram, to the left.

In the user interface, I have to create space for those three items. Each one is a `textOutput`, so I add these three lines to the user interface:

```
textOutput("mean"),
textOutput("median"),
textOutput("sd")
```

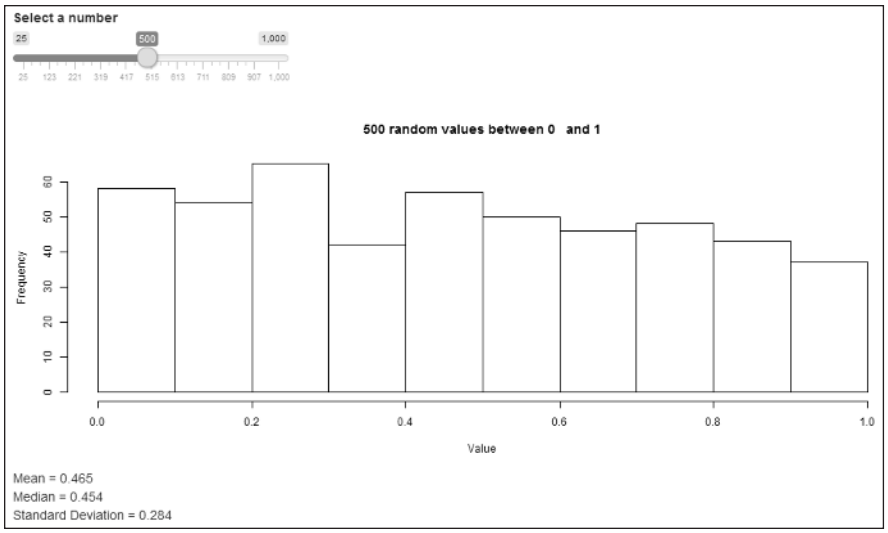

FIGURE 1-5: The shiny app with the mean, the median, and the standard deviation.

Of course, I also have to make changes to the server. Remember, I add

```
histdata <- reactive({
    runif(input$number,min=0,max=1)
    })
```

at the beginning of the server code.

For the `textOutputs`, I add

```
output$mean <- renderText({paste("Mean =",round(mean(histdata()),3)
  )
  })

output$median <- renderText({paste("Median =",round(median(histdata()),3)
  )
  })

  output$sd <- renderText({paste("Standard Deviation =",round(sd(histdata()),3)
  )
  })
```

The whole thing is

```
library(shiny)
ui <- fluidPage(
  sliderInput(inputId = "number",
              label = "Select a number",
```

```
              value = 500, min = 25, max = 1000),
  plotOutput("hist"),
  textOutput("mean"),

  textOutput("sd")
)
server <- function(input, output) {

  histdata <- reactive({
    runif(input$number,min=0,max=1)
    })

  output$hist <- renderPlot({
    hist(histdata(),xlab="Value",
    main=paste(input$number,"random values between 0   and 1")
  )
  })

  output$mean <- renderText({paste("Mean =",round(mean(histdata()),3)
  )
  })

  output$median <- renderText({paste("Median =",round(median(histdata()),3)
  )
  })

  output$sd <- renderText({paste("Standard Deviation =",round(sd(histdata()),3)
  )
  })
}
shinyApp(ui = ui, server = server)
```

Working with ggplot

By now, you likely know that I'm a huge fan of the `ggplot2` package. After reading this book, I hope you become one, too. In this section, I show you how to use `ggplot` functions to create the first version of the app from the preceding section. When it's done, it will look like Figure 1-6. (As in the preceding section, I show the app in an RStudio window. Click Open in Browser to see it in your browser.)

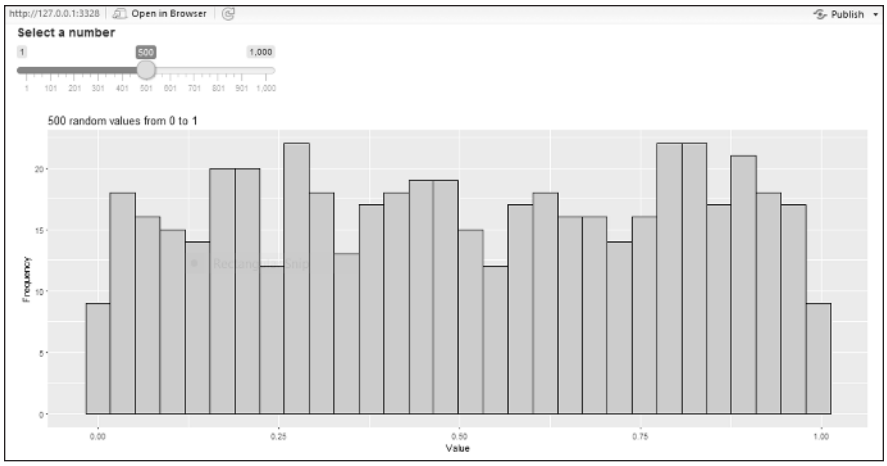

FIGURE 1-6: The first version of the `shiny` app from the preceding section, rendered in `ggplot2`.

To get started, I follow the steps in the preceding section to create a new application called `Uni formRandomggplot` in a new directory called `shinydir2`. Again, I delete the sample code and begin the coding with these two lines:

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

The user interface code remains the same as in the preceding section's first version:

```
ui <- fluidPage(
  sliderInput(inputId = "number",
              label = "Select a number",
              value = 500, min = 1, max = 1000),
  plotOutput("hist")
)
```

Changing the server

The function that does the plotting has to change. Instead of a base R function, I'm going to put `ggplot()` into `renderPlot()`'s curly brackets. Recall from Chapter 1 of Book 2 that `ggplot()` has to have a data frame as its first argument. So I can't just pass `runif(input$number, min = 0, max=1)` as an argument to `ggplot()`.

Instead, I have to turn the sample of `input$number` values into a data frame, and here's how I do it:

```
df <- data.frame(runif(input$number, min=0,max=1))
```

That would be the first line of code I put into `renderPlot()`'s curly brackets.

The second argument to `ggplot()` is `aes()`, which maps the values in the data frame into the *x*-axis of the histogram. This means I have to have a name for the column of values in the `df` data frame:

```
colnames(df)<-c("Value")
```

That's the second line of code in the curly brackets.

Now I can start on the plot:

```
ggplot(df,aes(x=Value))+
```

And I can add the histogram

```
geom_histogram(color = "black", fill = "grey80")+
```

and some landscaping:

```
labs(y="Frequency",title = paste(input$number,"random values from 0 to 1"))
```

Altogether, the code for the server looks like this:

```
server <- function(input, output) {
output$hist <- renderPlot({
    df <- data.frame(runif(input$number, min=0,max=1))
    colnames(df)<-c("Value")
    ggplot(df,aes(x=Value))+
             geom_histogram(color = "black",fill="grey80")+
             labs(y="Frequency",
title = paste(input$number,"random values from 0 to 1"))
    })
}
```

Remember to add

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

With the code for the user interface (including the two `library()` functions) and the server (and `shinyApp()`) saved in `shinydir2`, click the Run App button to produce what you see in Figure 1-5.

A few more changes

In the Console pane, this line appears each time you move the slider:

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

This indicates that R has taken a guess about how to render the appearance of the histogram. Specifically, R takes a shot at the binwidth — the width of each bar. Some modifications eliminate the guesswork.

First, I want to add a slider that enables the viewer to set the binwidth. To do that, I add the following lines to the user interface code, placing them between the first `sliderInput()` and `plotOutput()`:

```
sliderInput(inputId = "binwidth",
              label = "Select a binwidth",
              value = .05, min = .01, max = .10),
```

The first argument sets the identifier for this particular input, and the second puts a label above the slider. The third gives the starting binwidth, the fourth gives the minimum binwidth, and the fifth gives the maximum binwidth.

TIP

The (approximate) number of rendered bars is the range of values (1.00) divided by the selected binwidth. So the starting value (`.05`) produces 20 (ish) bars.

Changes to the `title` argument in the `labs()` function in the server add the binwidth information to the histogram title:

```
labs(y="Frequency",
title = paste(input$number,"random values from 0 to 1 with binwidth =",
   input$binwidth))
```

The whole megillah is shown here:

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

```
ui <- fluidPage(
  sliderInput(inputId = "number",
              label = "Select a number",
              value = 500, min = 1, max = 1000),
  sliderInput(inputId = "binwidth",
              label = "Select a binwidth",
              value = .05, min = .01, max = .10),
  plotOutput("hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
    df <- data.frame(runif(input$number, min=0,max=1))
    colnames(df)<-c("Value")
    ggplot(df,aes(x=Value))+
            geom_histogram(binwidth=input$binwidth,
                           color = "black",fill="grey80")+
            labs(y="Frequency",
title = paste(input$number,"random values from 0 to 1 with binwidth
   =",input$binwidth))
    })
}
shinyApp(ui = ui, server = server)
```

Press Ctrl+S to save it all in the `shinydir2` directory, and then run the app to produce the display in Figure 1-7.

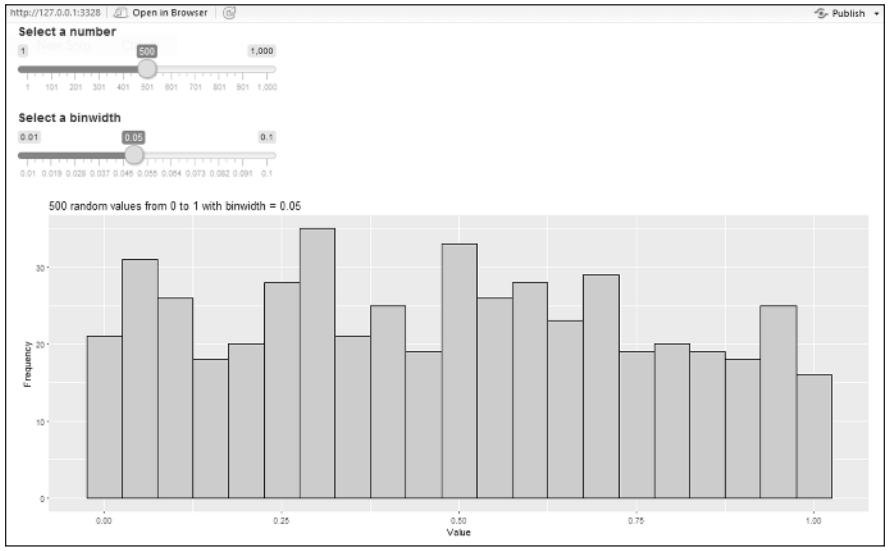

FIGURE 1-7: Adding a slider to enable the selection of binwidth.

Getting reactive with ggplot

To add the mean, median, and mode to the display you see in Figure 1-7 — the idea here is that it should match what you see in Figure 1-5 — I first add the `textOutputs` to the user interface, as before:

```
textOutput("mean"),
textOutput("median"),
textOutput("sd")
```

Things start to get a bit tricky in the server because I have to do two things: Use `reactive({})` to create a variable for `runif()`, and create a data frame for `ggplot()`. Why is this tricky? Because in the simpler version with just the plot and not the statistics, I was able to accomplish both at one time inside the reactive context of `renderPlot()`:

```
df <- data.frame(runif(input$number, min=0,max=1))
```

In this version, however, I have to create the variable in a reactive context outside `renderPlot({})` (so that I can use that variable to calculate the mean, median, and standard deviation) and the data frame inside `renderPlot({})` so that `ggplot()` can use it.

Here's the variable (`histdata`) in `reactive({})`:

```
server <- function(input, output) {
  histdata <- reactive({(runif(input$number, min=0,max=1))

  })
```

And here's the data frame (`df`) inside `renderPlot({})`:

```
  output$hist <- renderPlot({
    df <-data.frame(histdata())
    colnames(df)<-c("Value")
    ggplot(df,aes(x=Value))+
      geom_histogram(binwidth=input$binwidth, color = "black",fill="grey80")+
      labs(y="Frequency",
           title = paste(input$number,"random values from 0 to 1 with binwidth
                ="",input$binwidth))
```

And finally, here are the `output$`s:

```
output$mean <- renderText({paste("Mean =",round(mean(histdata()),3)
  )
  })

 output$median <- renderText({paste("Median =",round(median(histdata()),3)
  )
  })

 output$sd <- renderText({paste("Standard Deviation =",round(sd(histdata()),3)
 )
 })
```

TIP

Yes, I'm going to harp on this: Notice that after I define `histdata` in `reactive({})`, it's `histdata()` whenever I use it again.

HOW DOES ALL THIS WORK, REALLY?

When you drive, do you have to know the inner workings of your car's engine? Do you have to know exactly how your refrigerator keeps your food cold? If you answered yes to at least one of those questions, this sidebar is for you. Even if you didn't, you might still want to read it.

I hate to break this to you, boys and girls, but like computer animation, reactivity is an illusion. In computer animation, nothing moves across the screen: Instead, one pixel turns off, another turns on, and the illusion is that the pixel has moved from the first pixel's location to the second.

Likewise, in reactivity, it is *not* the case that the app only monitors the user and that when the user makes a change to the input (like moving the slider to change `input$number`), the output (like the plot in `output$hist`) changes accordingly. Instead, the server constantly recomputes everything in the app every few microseconds. So if the user moves the slider, for example (or changes the input in some other way), within microseconds the output updates.

Wait a (micro) second. Suppose the user doesn't make a change. Then what? Recomputation takes place anyway. It's just that everything recomputes its previous results, and it looks like the app hasn't changed at all. Bear in mind that whether or not the user does anything, recomputing is always going on in the background.

The illusion is that the user's action immediately causes the app's reaction. And, like computer animation, that's a pretty useful illusion!

Make those changes and run the app. It should look like Figure 1-8:

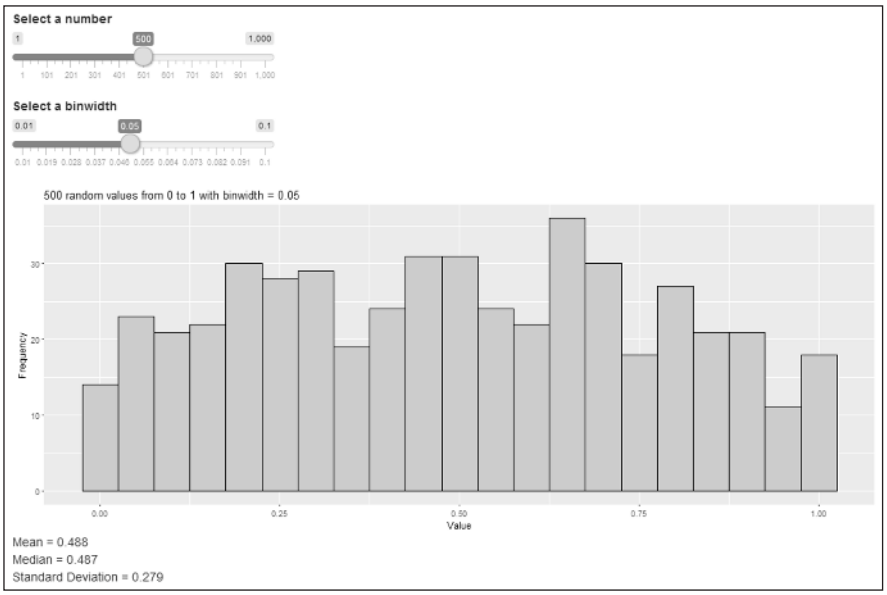

FIGURE 1-8: The `ggplot2` version of the first `shiny` app, with statistics added.

Another shiny Project

In this section, I move from a `shiny` app based on random sampling to an app based on data. The data that forms the basis of this project is in the data frame `airquality`, which lives in the `datasets` package.

As I mention in Chapter 2 of Book 2, this data frame holds data for temperature, wind velocity, solar radiation, and ozone for New York City for May–September 1973. To refresh your memory, here are the first six rows of the data:

```
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

The objective is an app that shows a scatterplot of two user-selected variables (excluding `Month` and `Day`) along with statistical summaries (correlation and regression) of the relationship between the variables. I show you how to create two versions: one in base R graphics and the other in `ggplot`.

The base R version

Figure 1-9 shows the finished product. The user selects an *x*-variable from one drop-down menu, and a *y*-variable from the other. The application then produces a scatterplot, which contains the regression line that summarizes the relationship between the two variables. The first line of the scatterplot title includes the selected variables. The second line shows the correlation coefficient (r) between the two, along with the equation of the regression line in the plot.

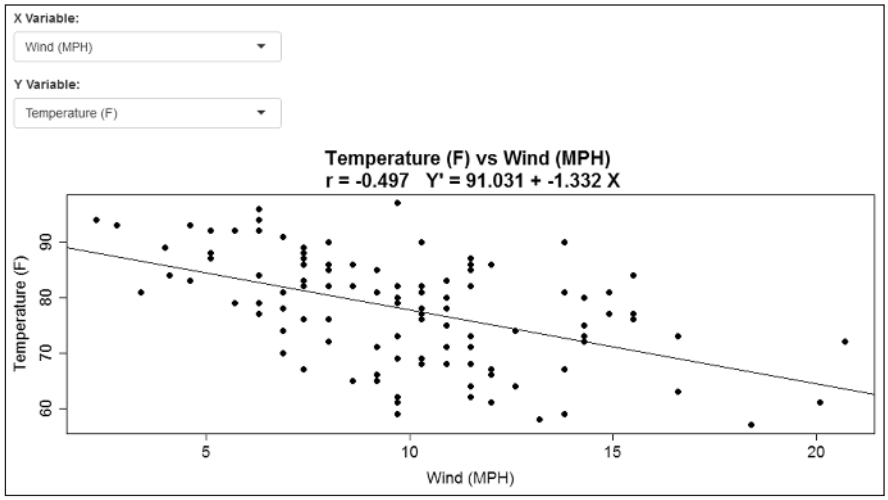

FIGURE 1-9: A `shiny` app for the `airquality` data frame.

I begin by choosing File➪NewFile➪Shiny Web App to create a new app called `AirQuality` in a new directory called `shinydir3`. I then delete the sample code.

For the new code, the first thing I do is to attach the library that contains the data frame:

```
library(datasets)
```

I'll have to clean up the data by eliminating missing values. The function that does that, `drop_na()`, lives in the `tidyr` package, so I add

```
library(tidyr)
```

I describe `drop_na()` in Chapter 2 of Book 1.

One more package, `tibble`, supplies a useful function called `rownames_to_column()`, which I also describe in Chapter 2 of Book 1. I use it here in a moment, so I add its package:

```
library(tibble)
```

Next, I delete the missing values from `airquality`:

```
aq.no.missing <-drop_na(airquality)
```

The newly created data frame `aq.no.missing` is the one I use going forward.

The next task is to provide a set of options for the *x*-variable menu and for the *y*-variable menu. The options, of course, are the same for both variables. I create the vector:

```
options <- c("Ozone (parts per billion)" = "Ozone",
             "Solar (Langleys)" = "Solar.R",
             "Wind (MPH)" = "Wind",
             "Temperature (F)" = "Temp")
```

Each term of the vector is a pair. The first element of each pair is the label that appears on the drop-down menu. The second element is the name of the variable in `aq.no.missing` that the first element connects to.


TECHNICAL STUFF

What is a langley? Used as a measure of solar radiation, one langley is one small calorie per square centimeter of irradiated area. What's a small calorie? The amount of energy required to raise 1 gram of water by 1 degree Celsius. (A thousand of them make up each calorie you count in food.) Aren't you glad you asked?

Take another look at Figure 1-7. Notice that the names in the plot title and on the axes are the labels from the drop-down menus, not variable names from the data frame. I think this makes the whole thing more informative. How do I get this done?

First, I turn the `options` vector into a data frame:

```
df.options <-data.frame(options)
```

Here's what that data frame looks like:

```
> df.options
                           options
Ozone (parts per billion)   Ozone
Solar (Langleys)          Solar.R
Wind (MPH)                   Wind
Temperature (F)              Temp
```

For this data frame to be useful, the row names on the left have to constitute a data column, so

```
df.lv <-rownames_to_column(df.options)
```

makes that happen. I use `lv` in the new data frame name to denote `label` (the name that appears on the menu) and `value` (the corresponding variable name in the data frame). To complete this data frame, I name its columns:

```
colnames(df.lv) <- c("label","value")
```

This data frame now looks like this:

```
> df.lv
                       label   value
1 Ozone (parts per billion)   Ozone
2          Solar (Langleys) Solar.R
3               Wind (MPH)    Wind
4           Temperature (F)    Temp
```

On to the user interface:

```
ui <- fluidPage(
  selectInput("X", "X Variable:",
              options),
  selectInput("Y", "Y Variable:",
              options),
  plotOutput("scatter")
)
```

Once again, it's a fluid page. Each `selectInput()` is a drop-down menu. The first argument is its name, the second argument is its onscreen label, and the third is the `options` vector that presents the choices. And `plotOutput()` sets aside the space for the plot.

Now for the server. The overall structure of the server, remember, is

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

The first item between the brackets assigns the user selections `input$X` and `input$Y` to a data frame I call `selections`. I do this in a reactive context (see the earlier section "Getting reactive"):

```
selections <- reactive({
    aq.no.missing[, c(input$X, input$Y)]
  })
```

REMEMBER

Here I go again: I've created `selections` in a reactive context (within `reactive({})`, in other words), and the next time I use it, I have to refer to it as `selections()`.

The comma within the square brackets means "all rows in the `aq.no.missing` data frame." The second expression `c(input$X, input$Y)` limits those rows to just the variables the user has selected. The result is that I can now refer to all rows in the first selected variable as

```
selections()[,1]
```

and to all the rows in the second as

```
selections()[,2]
```

which I will do almost immediately. Stay tuned.

The next item in the server is the `output` function, whose overall structure is

```
output$scatter <- renderPlot({})
```

The code for rendering the output goes between the curly brackets.

And now, as promised, I use those references to the two selected variables. I assign the first user selection to a variable called `x_column`:

```
x_column <- selections()[,1]
```

and the second to `y_column`:

```
y_column <- selections()[,2]
```

The correlation coefficient is

```
correlation <-cor(x_column,y_column)
```

and the regression is

```
regression <- lm(y_column ~ x_column)
```

To put the equation of the regression line into the title, I have to know its intercept (where the line meets the *y*-axis) and its slope (how slanted it is). In base R graphics, I also have to have those pieces of information to plot the regression line.

The result of a regression analysis is a list. For a regression analysis of `Temp` dependent on `Wind`, for example, part of that list looks like this:

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  91.0305     2.3489  38.754  < 2e-16 ***
Wind         -1.3318     0.2226  -5.983 2.84e-08 ***
---
```

To retrieve the intercept from the list, the expression is

```
intercept <- regression$coefficients[1]
```

And to retrieve the slope, it's

```
slope <- regression$coefficients[2]
```

Two more pieces of information and I'm ready to plot. So far, the R code has worked with the variable names that correspond to the user selections, like `Wind` and `Temp`. In the plot, remember, I want to use the names on the menus — `Wind (MPH)` and `Temperature (F)` — for the title and for the axis labels.

So I'm looking for the label names that correspond to the selected variable names. Here's where that `df.lv` data frame comes into play. For the label for the *x*-variable, I'm looking for

```
X_Label <- df.lv$label[whose corresponding df.lv$value matches input$X]
```

Fortunately, R provides a neat little trick that fills the bill. It's a function called `which()`, and here's how to use it:

```
X_Label <- df.lv$label[which(df.lv$value == input$X)]
```

And for the label for the *y*-variable, it's

```
Y_Label <- df.lv$label[which(df.lv$value == input$Y)]
```

And now, here's the `plot()` function:

```
plot(x=x_column,y=y_column,xlab = X_Label,ylab = Y_Label,
         cex.axis = 1.5,cex.lab = 1.5, pch = 20, cex = 2,
         main = paste(Y_Label,"vs",X_Label, "\n r =",round(correlation,3),"
            Y' =",round(intercept,3),"+",round(slope,3),"X"), cex.main=1.8)
```

The first two arguments, `x` and `y`, are the variables to plot. The next two, `xlab` and `ylab`, are the titles for the axes. The `cex.axis` argument specifies the size of the numbers on the axes, and `cex.lab` is the size of the axes labels. The value `1.5` means "1.5 times the normal size of a character." The next argument, `pch`, means that the plot character is a filled circle, and its size, `cex`, is 2.

The argument `main` is the title. I use `paste()` to put Y_Label and X_Label into the title. `\n` means to continue on the next line, where I paste the rounded `correlation` (rounded to three places) as well as the rounded `intercept` and the rounded `slope` into the regression equation. The size of the title, `cex.main`, is 1.8.

One more function draws the regression line:

```
abline(intercept,slope)
```

Here's the whole thing, including the `shinyApp()` function at the end:

```
library(datasets)
library(tidyr)
library(tibble)


aq.no.missing <-drop_na(airquality)

options <- c("Ozone (parts per billion)" = "Ozone",
             "Solar (Langleys)" = "Solar.R",
             "Wind (MPH)" = "Wind",
             "Temperature (F)" = "Temp")
df.options <-data.frame(options)
df.lv <-rownames_to_column(df.options)
colnames(df.lv) <- c("label","value")

ui <- fluidPage(
```

```
  selectInput("X", "X Variable:",
              options),

  selectInput("Y", "Y Variable:",
              options),

  plotOutput("scatter")


)
server <- function(input, output) {
  selections <- reactive({
    aq.no.missing[, c(input$X, input$Y)]

  })


  output$scatter <- renderPlot({

    x_column <- selections()[,1]
    y_column <- selections()[,2]

    correlation <-cor(x_column,y_column)
    regression <- lm(y_column ~ x_column)
    intercept <- regression$coefficients[1]
    slope <- regression$coefficients[2]

    X_Label <- df.lv$label[which(df.lv$value == input$X)]
    Y_Label <- df.lv$label[which(df.lv$value == input$Y)]

    plot(x=x_column,y=y_column,xlab = X_Label,ylab = Y_Label,
         cex.axis = 1.5,cex.lab = 1.5, pch = 20, cex = 2,
         main = paste(Y_Label,"vs",X_Label,
                      "\n r =",round(correlation,3),"
           Y' =",round(intercept,3),"+",round(slope,3),"X"),
         cex.main=1.8)
    abline(intercept,slope)
  })


}


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

Save the file and run the app!

The ggplot version

Rendered in `ggplot()`, this app looks like Figure 1-10.

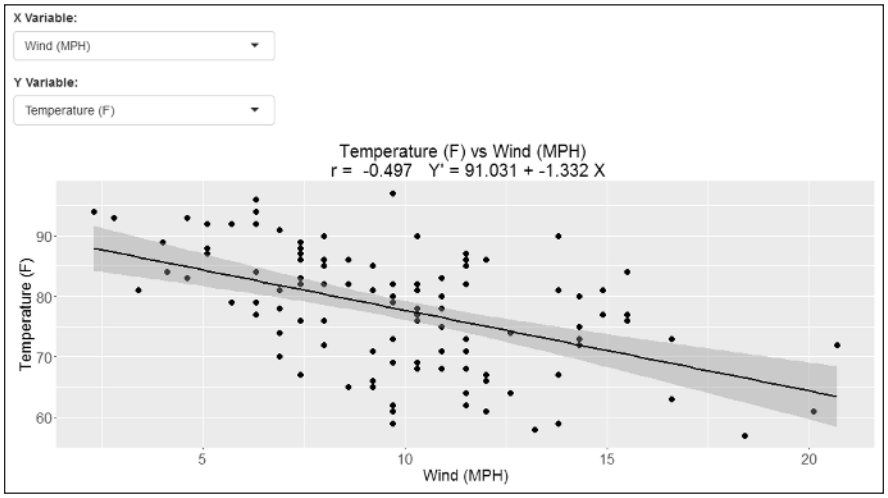

FIGURE 1-10: The app from the preceding section, rendered in ggplot.

The code is the same as in the base R version, except that I have to add

```
library(ggplot2)
```

to the beginning, and of course I have to change the plotting function in `output$scatter`.

Instead of `plot()`, I begin with `ggplot()`:

```
ggplot(selections(),aes(x=x_column,y=y_column))+
```

The first argument is the data frame that supplies the data; `aes()` then maps the first selected variable to `x` and the second selected variable to `y`.

Next, I add `geom_point()` to specify that I want points to appear in the plot:

```
geom_point(size=3) +
```

and the argument shows how big the points should be.

Adding a `labs()` function renders the *x*-axis, *y*-axis, and title:

```
    labs(x = X_Label,y = Y_Label,
           title = paste(Y_Label,"vs",X_Label,
           "\n r = ",round(correlation,3),"                        Y'
                ="round(intercept,3),"+",round(slope,3),"X"))+
```

To set the sizes of the fonts, I use a `theme()` function:

```
    theme(axis.title.x = element_text(size=18),
            axis.text.x = element_text(size=17),
            axis.title.y = element_text(size=18),
            axis.text.y = element_text(size=17),
            plot.title = element_text(hjust = 0.5,size=20))+
```

In `plot.title`, `hjust =0.5` centers the title.

Finally, `geom_smooth()` plots the regression line:

```
    geom_smooth(method="lm",col="black")
```

The first argument specifies a linear model (linear regression, in this example), and the second makes the line black. Notice that, unlike in base R, it's not necessary to specify the slope or the intercept.

The shadow around the regression line in Figure 1-10 represents the *standard error of estimate* — a measure of variability around the line. The tighter the shadow, the better the fit of the line to the data. (Note what happens when the *x*-variable and the *y*-variable are the same.). To eliminate the shadow, add `se=FALSE` as an argument to `geom_smooth()`.

Here's the entire set of functions for the `ggplot` version:

```
ggplot(selections(),aes(x = x_column,y = y_column))+
      geom_point(size=3) +
      labs(x = X_Label,y = Y_Label,
           title = paste(Y_Label,"vs",X_Label,
           "\n r = ",round(correlation,3),"
           Y' =",round(intercept,3),"+",round(slope,3),"X"))+
      theme(axis.title.x = element_text(size=18),
            axis.text.x = element_text(size=17),
            axis.title.y = element_text(size=18),
            axis.text.y = element_text(size=17),
            plot.title = element_text(hjust = 0.5,size=20))+
      geom_smooth(method="lm",col="black")
```

Substitute this set of functions for the `plot()` function and `abline()` in the base R version, save, and run the application.

Suggested Project

Feeling adventurous? Take what you learned in this last project and try it out on a different data frame. It's a great way to build up your skill set.

I suggest `Cars93`, which lives in the `MASS` package. This data frame provides information on a number of variables (way more than four!) for 93 models of cars from 1993.

Good luck!