2

Building Your First Application

In the previous chapter we've looked at R, learned some of its basic syntax, and seen some examples of the power and flexibility that R and Shiny offer. This chapter introduces the basics of Shiny. In this chapter we're going to build our own application to interactively query results from the Google Analytics API. We will cover the following topics:

- Basic structure of a Shiny program
- Selection of simple input widgets (checkboxes and radio buttons)
- Selection of simple output types (rendering plots and returning text)
- Selection of simple layout types (page with sidebar and tabbed output panel)
- Handling reactivity in Shiny

Program structure

In this chapter, in just a few pages, we're going to go from the absolute basics of building a program to interactively query data downloaded from the Google Analytics API. Let's get started by having a look at a minimal example of a Shiny program. The first thing to note is that Shiny programs are the easiest to build and understand using two scripts, which are kept within the same folder. They should be named `server.R` and `ui.R`. Throughout this book, all code will have a commented `server.R` and `ui.R` header to indicate which code goes in which file.

ui.R of minimal example

The `ui.R` file is a description of the UI and is often the shortest and simplest part of a Shiny application. Note the use of the `#` character, which marks lines of code as comments that will not be run, but which are for the benefit of humans producing the code:

```
####################################
##### minimal example - ui.R  #####
####################################

library(shiny) # load shiny at beginning at both scripts

shinyUI(pageWithSidebar( # standard shiny layout, controls on the
                                          # left, output on the right

  headerPanel("Minimal example"), # give the interface a title
  sidebarPanel( # all the UI controls go in here

    textInput(inputId = "comment",  # this is the name of the
                                    # variable- this will be
                                    # passed to server.R

              label = "Say something?", # display label for the
                                        # variable

              value = "" # initial value
    )
  ),

  mainPanel( # all of the output elements go in here
    h3("This is you saying it"), # title with HTML helper
    textOutput("textDisplay")  # this is the name of the output
                               # element as defined in server.R
  )
))
```


To run a Shiny program on your local machine you just need to do the following:

1. Make sure that `server.R` and `ui.R` are in the same folder.
2. Make this the R's working directory (using the `setwd()` command, for example `setwd("~/shinyFiles/minimalExample")`).
3. Load the Shiny package (`library(shiny)`).
4. Type `runApp()` at the console.

`runApp()` with the name of a directory within works just as well, for example, `runApp("~/shinyFiles/minimalExample")`. Just remember that it is a directory and not a file that you need to point to.

Let's have a detailed look at the file. We open by loading the Shiny package. You should always do that in both `server.R` and `ui.R` files. The first instruction, `shinyUI(pageWithSidebar(...` tells Shiny that we are using the vanilla UI layout, which places all the controls on the left-hand side and gives you a large space on the right-hand side to include graphs, tables, and text. All of the UI elements are defined within this instruction.

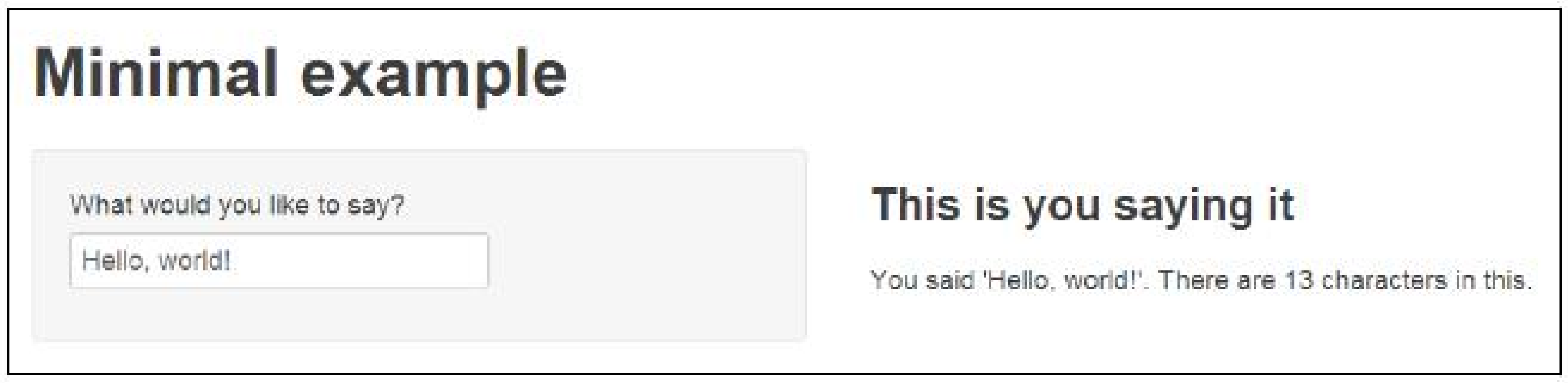
The next line, `headerPanel()`, gives the application a title. The next two instructions perform the main UI setup, with `sidebarPanel()` setting up the application controls and `mainPanel()` setting up the output area. `sidebarPanel()` will usually contain all of the input widgets, in this case there is only one: `textInput()`. `textInput()` is a simple widget that collects text from a textbox that users can interact with using the keyboard. The arguments are pretty typical among most of the widgets and are as follows:

- `inputId`: This argument names the variable so it can be referred to in the `server.R` file
- `label`: This argument gives a label to attach to the input so users know what it does
- `value`: This argument gives the initial value to the widget when it is set up—all the widgets have sensible defaults for this argument, in this case, it is a blank string, `""`

When you are starting out, it can be a good idea to spell out the default arguments in your code until you get used to which function contains which arguments. It also makes your code more readable and reminds you what the return value of the function is (for example, `value = TRUE` would suggest a Boolean return).

The final function is `mainPanel()`, which sets up the output window. You can see I have used one of the HTML helper functions to make a little title `h3("...")`. There are several of these functions designed to generate HTML to go straight on the page; type `?p` at the console for the complete list. The other element that goes in `mainPanel()` is an area for handling reactive text generated within the `server.R` file—that is, a call to `textOutput()` with the name of the output as defined in `server.R`, in this case, `"textDisplay"`.

The finished interface looks similar to the following screenshot:



If you're getting a little bit lost, don't worry. Basically Shiny is just setting up a framework of named input and output elements; the input elements are defined in `ui.R` and processed by `server.R`, which then sends them back to `ui.R` that knows where they all go and what types of output they are.

server.R of minimal example

Let's look now at `server.R` where it should all become clear:

```
######################################
##### minimal example - server.R #####
######################################

library(shiny) # load shiny at beginning at both scripts

shinyServer(function(input, output) { # server is defined within
                                              # these parentheses

  output$textDisplay <- renderText({ # mark function as reactive
                                              # and assign to
                                              # output$textDisplay for
                                              # passing to ui.R

   paste0("You said '", input$comment,           # from the text
          "'. There are ", nchar(input$comment), # input control as
" characters in this."                           # defined in ui.R
   )
  })
})
```

Let's go through line by line again. We can see again that the package is loaded first using `library(shiny)`. Note that any data read instructions or data processing that just needs to be done once, will also go in this first section (we'll see more about this as we go through the book). `shinyServer(...{...})` defines the bit of Shiny that's going to handle all the data. On the whole, two types of things go in here. Reactive objects (for example, data) are defined, which are then passed around as needed (for example, to different output instructions), and outputs are defined, such as graphs. This simple example contains only the latter. We'll see an example of the first type in the next example.

An output element is defined next with `output$textDsiplay <- renderText({..})`. This instruction does two basic things: firstly, it gives the output a name (`textDisplay`) so it can be referenced in `ui.R` (you can see it in the last part of `ui.R`). Secondly, it tells Shiny that the content contained within is reactive (that is, to be updated when its inputs changes) and that it takes the form of text. We cover advanced concepts in reactive programming with Shiny in a later chapter. There are many excellent illustrations of reactive programming at the Shiny tutorial pages `http://rstudio.github.io/shiny/tutorial/#reactivity-overview`.

The actual processing is very simple in this example. Inputs are read from `ui.R` by the use of `input$...`, so the element named in `ui.R` as `comment` (go and have a look at `ui.R` now to find it) is referenced with `input$comment`.

The whole command uses `paste0()` to link strings with no spaces (equivalent to `paste(..., sep = "")`), picks up the text the user inputted with `input$comment`, and prints it along with the number of characters within it (`nchar()`) and some explanatory text.

That's it! Your first Shiny application is ready. Using these very simple building blocks you can actually make some really useful and engaging applications.

Optional exercise

If you want to have a practice before we move on, take the existing code and modify it so that the output is a plot of a user-defined number of observations, with the text as the title of the plot. The plot call should look like the following:

```
hist(rnorm(XXXX), main = "YYYY")
```

In the preceding line of code `XXXX` is a number taken from a function in `ui.R` that you will add (`sliderInput()` or `numericInput()`) and `YYYY` is the text output we already used in the minimal example. You will also need to make use of `renderPlot()`, type `?renderPlot` in the console for more details.

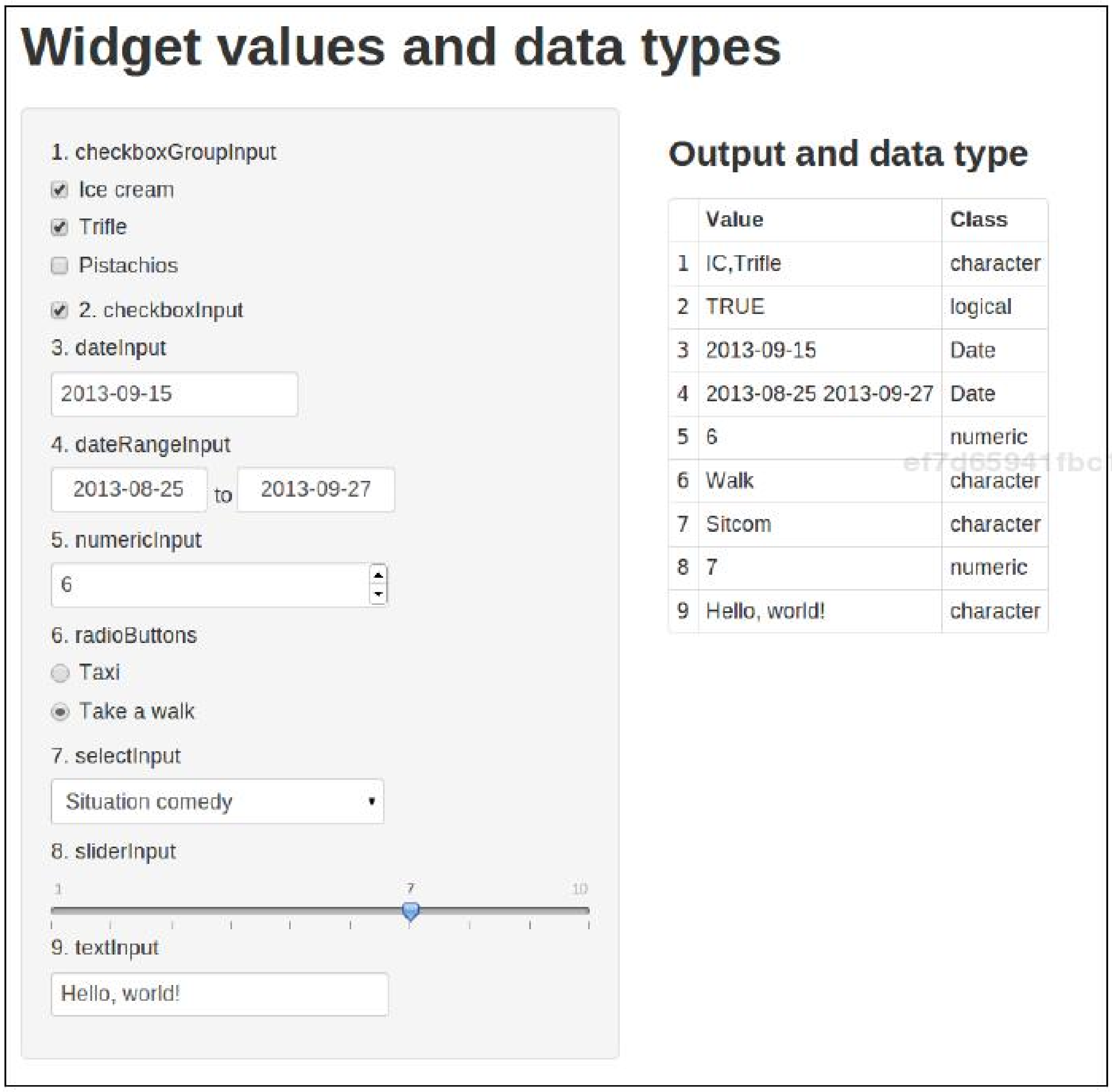
So far in this chapter we have looked at a minimal example, learned about the basic commands that go in the `server.R` and `ui.R` files. Thinking about what we've done in terms of reactivity, the `ui.R` file defines a reactive value, `input$comment`. The `server.R` file defines a reactive expression, `renderText()`, that depends on `input$comment`. Note that this dependence is defined automatically by Shiny. `renderText()` uses an output from `input$comment`, so Shiny automatically connects them. Whenever `input$comment` changes, `renderText()` will automatically run with the new value. The extra credit exercise gave two reactive values to the `renderPlot()` call, and so, whenever either changes, `renderPlot()` will rerun. In the rest of this chapter we will look at an application that uses some slightly more advanced reactivity concepts, and by the end of the book, we will have covered all the possibilities that Shiny offers and when to use them.

Widget types

Before we move on to a more advanced application, let's have a look at the main widgets that you will make use of within Shiny. I've built a Shiny application that will show you what they all look like, as well as showing their outputs and the type of data they return. To run it, just enter the following command:

```
> runGist(6571951)
```

This is one of several built-in functions of Shiny that allow you to run code hosted on the Internet. Details about sharing your own creations and other ways are discussed in *Chapter 5*, *Running and Sharing Your Creations*. The finished application looks like the following:



You can see the function names (**checkboxGroupInput** and **checkboxInput**) as numbered entries on the left-hand side panel; for more details, just type `?checkboxGroupInput` at the console.

If you're curious about the code, it's available at `https://gist.github.com/ChrisBeeley/6571951`.

Google Analytics application

Now that we've got the basics, let's build something useful. We're going to build an application that allows you to interactively query data from the Google Analytics API. There is no room within this book to discuss registering for and using the Google Analytics API; however, you will very likely wish to make use of the wonderful **rga** package if you want to get your own Analytics data into R. This package provides an interface between the API and R; at the time of writing, it is still in development and cannot be downloaded using standard package management. Instructions for downloading, installing, and using rga can be found at `https://github.com/skardhamar/rga`.

To keep things simple, we will concentrate on data from a website that I worked on. We'll also use a saved copy of the data that is loaded into the application the first time it runs. A full production of the application could obviously query the API every time it launched or on a daily or weekly basis, depending on how many users you expected (the API limits the number of daily queries from each application). Note that we would *not* query the API as part of a reactive expression unless there was a clear need for the application to be constantly up-to-date, because it would use a lot of the allocated queries, as well as making the program run a lot more slowly. In practice, this means the query, just like the data load function used in the following code, would be given at the top of the `server.R` file, outside of the call to `shinyServer({...})`. It will be launched each time the application is run (or it is trivially simple to write code that ensures this only occurs once per day with the results stored until the application is launched on the next day).

If you like any of the analysis that we come up with or want to extend it, you can always import your own Analytics data and load it in, as here, or query the API online if you want the application to be simple for others to use. All the data and code is hosted on GitHub and can be downloaded from `http://github.com/ChrisBeeley/GoogleAnalytics`.

The UI

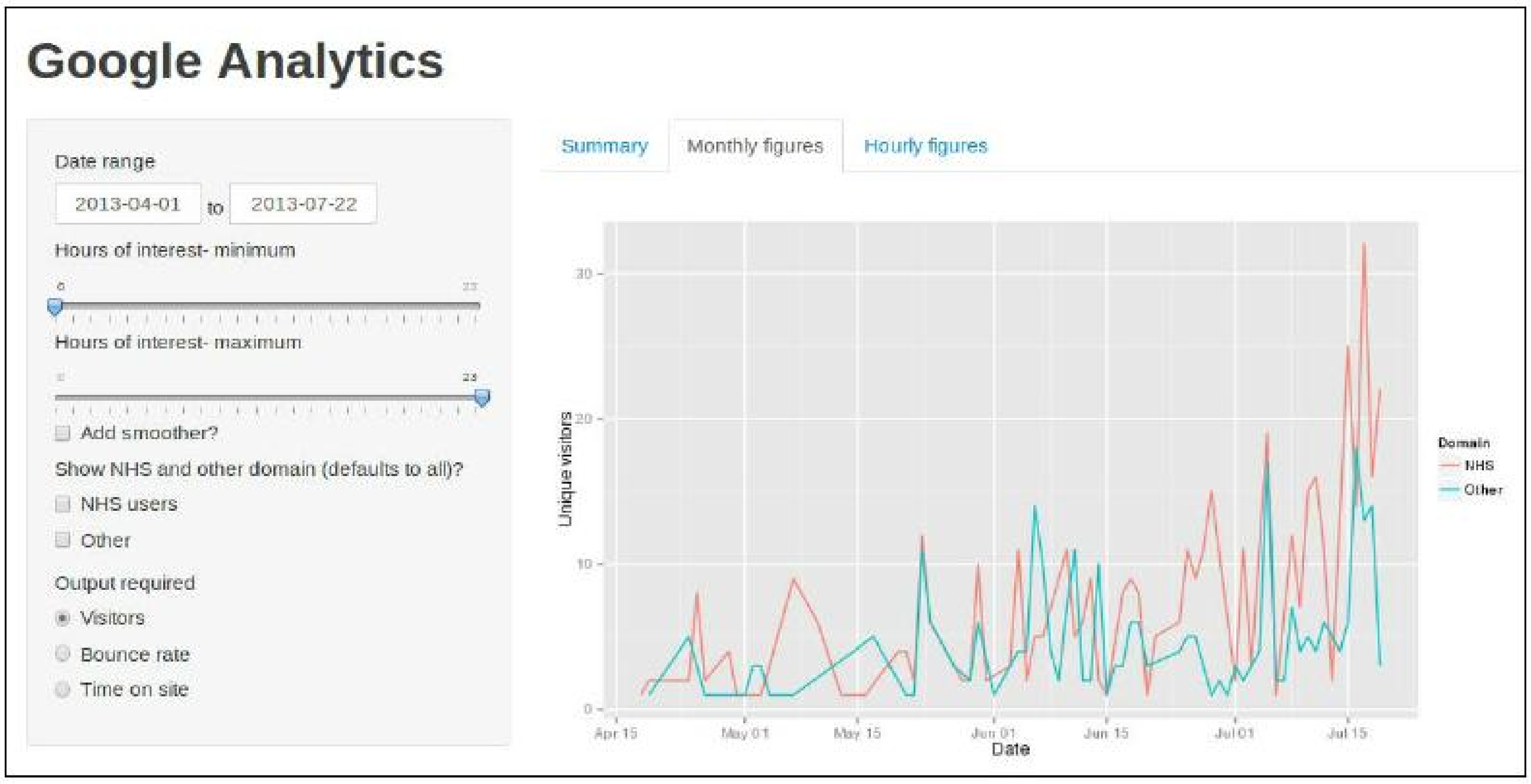
If you can, download and run the code and data (the data goes in the same folder as the code) so you can get an idea of what everything does. If you want to run the program without copying the actual data and code to your computer (copying data and code is preferable, so you can play with it), just use another function for sharing and running applications (we will discuss this in *Chapter 5*, *Running and Sharing Your Creations*):

```
> runGitHub("GoogleAnalytics", "ChrisBeeley")
```

In simple terms, the program allows you to select a date and time range and then view a text summary, or a plot of monthly or hourly figures. There are three tabbed windows in the output region where users can select the type of output they want (**Summary**, **Monthly figures**, and **Hourly figures**).

The data is from a health service (known locally as NHS) website, so users might be interested to show data that originates from domains within the NHS and compare it with data that originates from all other domains. There is an option to add a smoothed line to the graph, and three types of data are available: number of unique visitors, bounce rate (how many users leave the site after the first page they land on), and the average amount of time users spend on the site.

The following screenshot shows it in action:



As in many Shiny applications, `ui.R` is by far the simpler of the two code files and is as follows:

```
###################################
##### Google Analytics - ui.R #####
###################################

library(shiny)

shinyUI(pageWithSidebar(

  headerPanel("Google Analytics"),
```

```
    sidebarPanel(

      dateRangeInput(inputId = "dateRange",
                     label = "Date range",
                     start = "2013-04-01",
                     max = Sys.Date()
      ),
```

`dateRangeInput()` gives you two nice date widgets for the user to select a start and end point. As you can see, it's given a name and a label as usual; you can specify the start and end date (as done here, don't use the default behavior which gives the current system date) as well as a maximum date (manually given `Sys.Date()`, that is the system date, as used in this case). There are a lot of other ways to customize, such as the way the date is displayed in the browser, whether the view defaults to months, years, or decades, and others. Type `?dateRangeInput` in the console for more information:


```
      sliderInput(inputId = "minimumTime",
                  label = "Hours of interest- minimum",
                  min = 0,
                  max = 23,
                  value = 0,
                  step = 1),
```

`sliderInput()`, used in the extra credit exercise in this chapter, gives you a graphical slider that can be used to select numbers. Here the minimum, maximum, initial value, and step between values are all set (0 and 23 hours, with a step of 1, which is how Google Analytics returns the `hour` variable); again, for more details type `?sliderInput` in the console:

```
      sliderInput(inputId = "maximumTime",
                  label = "Hours of interest- maximum",
                  min = 0,
                  max = 23,
                  value = 23,
                  step = 1),

      checkboxInput(inputId = "smoother",
                    label = "Add smoother?",
                    value = FALSE),
```

checkboxInput() very simply gives you a tick box that returns TRUE when ticked and FALSE when unticked. This example includes all the possible arguments, giving it a name and label and selecting the initial value:

```
checkboxGroupInput(inputId = "domainShow",
                   label = "Show NHS and other domain
                            (defaults to all)?",
                   choices = list("NHS users" = "NHS",
                                  "Other" = "Other")
                   ),
```

checkboxGroupInput() returns several checkboxes and is useful when users need to make multiple selections. Of note in this example is the use of a list to specify the options. This allows the display value (given to the user on the UI) and the return value (given to R for processing) to be different. Note the way elements in a list are named; it's quite a simple syntax: list("First name" = "returnValue1", "Second name" = "returnValue2"). You can see that this allows nicely formatted labels (with spaces in natural English) to be used in the label and computer-speak (camel case variable names with no spaces) to be used in the return value:


```
radioButtons(inputId = "outputType",
             label = "Output required",
             choices = list("Visitors" = "visitors",
                            "Bounce rate" = "bounceRate",
                            "Time on site" = "timeOnSite"))
```

radioButtons(), amazingly, will give you radio buttons. This allows the selection of one thing and one thing only from a list. Again, because a named list is used, an optional (...selected = ...) argument can be used to determine the default selection, otherwise the first value is used as the default:

```
    ),
    mainPanel(
      tabsetPanel(
        tabPanel("Summary", textOutput("textDisplay")),
        tabPanel("Monthly figures", plotOutput("monthGraph")),
        tabPanel("Hourly figures", plotOutput("hourGraph"))
      )
    )
  ))
```

Probably the most unfamiliar part of this code is the use of `tabsetPanel()`. This allows multiple frames of output to be shown on the screen and selected by the user, as is common in GUIs that support tabbed frames. Note that processing is only carried out for the currently selected tab; invisible tabs are not updated behind the scenes but rather when they are made active. This is useful to know where some or all tabs require significant data processing.

The setup is very simple, with a call to `tabsetPanel()` containing several calls to `tabPanel()` in which each of the tabs is defined with a heading and a piece of output, as defined in `server.R`.

Data processing

As you write more and more complex programs, it's the `server.R` file that will become the largest because this is where all the data processing and output goes on, and even where some of the functions that handle advanced UI features live. Instead of going through all of the code line by line, as we did before, we're going to look at the chunks in order and talk about the kinds of things that are done in each section in typical Shiny applications.

The first chunk of code looks like the following:

```
#######################################
#### Google Analytics - server.R #####
#######################################

library(shiny)
library(plyr)
library(ggplot2)

load("analytics.Rdata") # load the dataframe
```


This chunk is run once every time the application is launched. This is where all the data preparation will take place. In this example, it is very simple, and once the relevant R packages are loaded, the whole dataframe is loaded in ready for use. Sometimes you will be able to do all of your data processing "offline" and load the data in, being fully prepared in this way. Sometimes, however, you may rely on a spreadsheet that changes on the server regularly, or, as in this case, you may wish to query the Google API. In cases like these, this is the place to do the data cleaning and preparation necessary to run the R code with the dataset. The code to do that is outside the scope of this section, but it suffices to say that as you get more confident with R, you will be analyzing more and more complex datasets and you will find it useful to do more data preparation within this section.

Reactive objects

The next section is contained within the reactive part contained within the `shinyServer({...})` call. Up until now this section has just contained a list of output commands that produce the output ready to fill the allocated spaces in `ui.R`. In the next chunk we're going to look at another way of managing your analysis. Sometimes you want to prepare a reactive dataset once and then pass it around the program as needed. This might be because you have tabbed output windows (as in this case) that use the same dataset and you don't want to write and maintain code that prepares the data according to the values of reactive inputs within all three functions. There are other times when you want to control the processing of data because it is time-intensive or it might make an online query (such as in the case of a "live" Google Analytics application that queries data live in response to reactive inputs). The way that you can take more control over data processing from reactive inputs, rather than distributing it through your output code, is to use reactive objects. A **reactive object**, like a reactive function, changes when its input changes. Unlike a reactive function, it doesn't do anything, but is just a data object (dataframe, number, list, and so on) that can be accessed by other functions. Let's have a look at an example:


```
# prep data once and then pass around the program

passData <- reactive({
```

Some of the R code will be a little unfamiliar to you, but for now just concentrate on what the program is actually doing. The first thing to note is that, unlike previous examples, we are not making a call such as `output$lineGraph <- renderPlot({...})` or `output$summaryText <- renderText({...})`. Instead, we are marking whatever is inside the call as *reactive* by enclosing it in `reactive({...})`. This generates a reactive object called `passData`. This can be accessed just like any other dataframe like this: `passData()` (for the whole dataframe) or `passData()$variableName` (for a variable), or `passData()[, 2:10]` (for the second to the tenth variable). Note the brackets after `passData`.


```
  analytics <- analytics[analytics$Date %in%
                  seq.Date(input$dateRange[1],
                  input$dateRange[2], by = "days"),]
```

This command selects the dates that the user is interested in using the vector of two dates within `input$dateRange` as defined in `ui.R`. Note that the first of these dates is selected with `input$dateRange[1]` and the second with `input$dateRange[2]`. One of the nice things about this widget is it ensures users can only select logical values, that is, they can only select start dates that occur before end dates, and end dates that occur after start dates. Have a go and see. This keeps your code simpler because you know that only valid values will be returned (selection of the same date is possible, so your code will need to handle that case):

```
    analytics <- analytics[analytics$Hour %in%
      as.numeric(input$minimumTime):
      as.numeric(input$maximumTime),]
```


This next instruction restricts the data to the requested time period. In this case, the user can select "wrong" values for the minimum and maximum time, with the maximum value being lower than the minimum value, because this part is made of two separate widgets. It doesn't affect the data extract in this case; the code will just match the sequence `10, 9, 8, 7, 6`, instead of the sequence `6, 7, 8, 9, 10`, so the data object is exactly the same. In a different application, you may need to check for the validity of the input or control the UI, so invalid selections are not possible. We will discuss the second possibility later on in the book:

```
  if(class(input$domainShow)=="character") {
    analytics <- analytics[analytics$Domain %in%
      unlist(input$domainShow),]

  }
```

And finally, the last statement restricts the data to either the NHS domain or the non-NHS domain according to user preference. The `if(){...}` statement checks to see if the user has made a selection before it subsets the data (an empty selection returns `NULL`, whereas any other selection will return an object of class character – a string, so that's what the code checks for). Quite often you will have to make sure that your code works with all the return types of the UI, or checks for valid input, whichever makes for the cleanest and simplest code:


```
    analytics

  })
```

We finish with the simple `analytics` instruction, which simply means "give `passData` the object **analytics**, which we've now defined as reactive based on the inputs in this instruction".

Outputs

Finally, the outputs are defined. Let's look first at the code that produces the first tab of output, monthly totals:

```
output$monthGraph <- renderPlot({

  graphData <- ddply(passData(), .(Domain, Date), numcolwise(sum))
```

The first instruction prepares the data using the user contributed package **plyr** (as with ggplot2, we have Hadley Wickham to thank for this package). This package is incredibly useful, but can be a little hard to understand at first. For now, just note that this instruction takes a dataframe as an input and then produces column sums based on unique combinations of domain and date. In this case, this means summing over the hours for each date or summing over the dates for each hour. This is the monthly graph, so we need to sum over the hours for each date. Instead of having 1 A.M. on the 21st, 2 A.M. on the 21st, 3 A.M. on the 21st, and so on, we just add them all up and have the totals for the 21st, the 22nd, and so on:


```
    if(input$outputType == "visitors"){

    theGraph <- ggplot(graphData,
      aes(x = Date, y = visitors, group = Domain, colour = Domain))
      + geom_line() + ylab("Unique visitors")

  }

  if(input$outputType == "bounceRate"){

    theGraph <- ggplot(graphData,
      aes(x = Date, y = bounces / visits * 100, group = Domain,
      colour = Domain)) +
      geom_line() + ylab("Bounce rate %")

  }

  if(input$outputType == "timeOnSite"){

    theGraph <- ggplot(graphData,
      aes(x = Date, y = timeOnSite / visits, group = Domain,
      colour = Domain)) +
      geom_line() + ylab("Average time on site")

  }
```

Following this we have three `if({...})` statements that correspond to the three values of the radio button: total visitors, bounce rate, and time on site. Each of these instructions sets up a ggplot graph with the same parameters:

- Date on the x axis
- Grouping variable distinguished by color for the domain (this will draw a separate line for each domain and color them differently)
- A call to `geom_line()` to tell ggplot that we want a line graph

The only parameter that changes is `y`. This is the variable that will be shown on the graph, and is either equal simply to visitors (that is, number of `visitors`), `bounces / visits * 100` (that is, the percentage of visitors who leave after the first page), or `timeOnSite / visits` (that is, the total time on site divided by the number of visits, to give the mean time on the site) and the `ylab("...")` argument that labels the y axis appropriately. Note that we have still not printed the graph anywhere, we are just setting it up.


```
    if(input$smoother){

      theGraph <- theGraph + geom_smooth()

    }
```

We can go on with the setup now with this instruction that checks to see if the user requested a smoothing line and, if they did, add one to the graph with `geom_smooth()`.

```
    print(theGraph)

  })
```

Finally, we give the instruction to `print()` the graph. This is always necessary in ggplot-based graphics in Shiny, whether you have built up the graph in a separate variable or just given one instruction on one line. Don't forget this! It's a very common cause of problems, as you will no doubt notice if you spend any time on forums or mailing lists. Many newcomers make this mistake. The hourly graph is built up and outputted in exactly the same way, except using the `hour` variable in place of the `date` variable:


```
  output$hourGraph <- renderPlot({

    graphData = ddply(passData(), .(Domain, Hour), numcolwise(sum))

    if(input$outputType == "visitors"){
```

```
    theGraph <- ggplot(graphData,
      aes(x = Hour, y = visitors, group = Domain,
      colour = Domain)) +
      geom_line() + ylab("Unique visitors")

  }

  if(input$outputType == "bounceRate"){

    theGraph <- ggplot(graphData,
       aes(x = Hour, y = bounces / visits * 100, group = Domain,
       colour = Domain)) +
       geom_line() + ylab("Bounce rate %")
```



```
  }

  if(input$outputType == "timeOnSite"){

    theGraph <- ggplot(graphData,
      aes(x = Hour, y = timeOnSite / visits, group = Domain,
      colour = Domain)) +
      geom_line() + ylab("Average time on site")

  }

  if(input$smoother){

    theGraph <- theGraph + geom_smooth()

  }

  print(theGraph)
```



```
})
```

Finally, the component that handles the text output runs as follows:

```
output$textDisplay <- renderText({
paste(
  length(seq.Date(input$dateRange[1], input$dateRange[2],
         by = "days")),
    " days are summarised. There were", sum(passData()$visitors),
    "visitors in this time period."
  )
})
```

You will be familiar with the `paste()` command by now; the first function within the `paste()` call produces a vector of dates between the two specified in the UI and then finds its length using, unsurprisingly, the `length()` command.

A note on the application code

Please note that, as at many points in this book, some of the decisions made around the `server.R` file were made to keep the code understandable and would not be used in a full application. The monthly and hourly graphics were drawn separately and each contained a data processing instruction at the beginning. A full application would do neither of these things. All data processing would be done in the first reactive call—producing either a list of two dataframes, one for each, or one larger frame that would feature values for both datasets. This makes the code easier to understand and maintain.

Further, although we have included two separate graph instructions and put them in different tabs, a full application would use one set of code that could handle both examples. Doing this requires a moderate level of proficiency with R. The code will be shorter, clearer, and easier to maintain, but more difficult to read and understand here if you are new to R.

Optional exercise

The Google Analytics application is reasonably intuitive and well featured (it doesn't, admittedly, compare all that favorably with Google's own offering!). However, as with the `server.R` code, some of the decisions around the UI setup were made for simplicity for the purposes of this book, to avoid flooding you with new widgets and ways of handling inputs in the second chapter. You may like to pause here and take a bit of time to update the code with some of the other UI elements Shiny offers to make the application function a bit more intuitive. Have a browse through the documentation yourself (`?shiny`) or make use of the following:

- `numericInput()`: This function gives both a textbox and a selection box to allow users to select a numeric value.
- `selectInput()`: This function allows a user to select one or multiple items from a list.
- `textInput()`: This function is not that useful in this case, but you can have some fun parsing its output with `as.numeric()` and using that as a numerical input

You will need to look at the return types for each of the widgets and make sure that the `server.R` code will accept them and, if not, change the code so that it will.

Those with some experience with R will no doubt be itching to fix the `server.R` file to clear up the issues outlined in the previous section. This will mainly sharpen your R skills and will also give you practice in some of the basics of scoping, classing, and passing data in a Shiny application. So if you feel up to it, have a go with this code too.

Summary

In this chapter we have covered a lot of ground. We've seen that Shiny applications are generally made up of two files: `server.R` and `ui.R`. We've learned what each part of the code does, including setting up `ui.R` with the position and type of inputs and outputs, and `server.R` with the data processing functions, outputs, and any reactive objects that are required.

The optional exercises have given you a chance to experiment with the code files in this chapter, varying the output types, using different widgets, and reviewing and adjusting their return values as appropriate. In addition, we've learned about the default layouts in Shiny, `pageWithSidebar()`, `mainPanel()`, and `tabsetPanel()`.

We've also learned about reactive objects and discussed when you might use reactive objects. There's more on finely controlling reactivity later in the book.

In the next chapter we're going to learn how to integrate Shiny with your own content, using HTML, CSS, and JavaScript.