3
Building Your Own Web Pages with Shiny

So, we've built our own application to query our site's data on Google Analytics. We've learned about the basic setup of a Shiny application and seen a lot of the widgets. It will be important to remember the majority of this basic structure because we are going to cover a lot of different territories in this chapter and, as a consequence, we won't have a single application at the end, like we did in the previous chapter. Instead, we will have lots of bits and pieces that you can use to start building your own content. Building one application with all of these different concepts would create several pages of code and it would be difficult to understand which part does what. As you go through the chapter, you might want to rebuild the Google Analytics application, or another of your own if you have one, using each of the concepts. If you do this, by the end you will have a beautifully styled and interactive application that you really understand. Or you might like to just browse through and pick out the things that you are particularly interested in; you should be able to understand each section on its own. Let's get started now. We are going to cover the following areas:

- Customizing Shiny applications, or whole web pages, using HTML
- Styling your Shiny application using CSS
- Turbo-charging your Shiny application with JavaScript and jQuery

Running the applications and code

For convenience, I have gathered together all the applications in this chapter. The link to the live versions as well as source code and data on my website can be found at `http://chrisbeeley.net/website/shinybook.html`. If you can, run the live version first, and then browse the code as you go through each example.

Shiny and HTML

It might seem quite intimidating to customize the HTML in a Shiny application, and you may feel that by going *under the hood*, it would be easy to break the application or ruin the styling. You may not want to bother rewriting every widget and output in HTML just to make one minor change to the interface.

In reality, Shiny is very accommodating, and you will find that it will quite happily accept a mix of Shiny and HTML code produced by you using Shiny helper functions, and the raw HTML written by you. So you can style just one button, or completely build the interface from scratch and integrate it with some other content. I'll show you all of these methods and provide some hints about the type of things you might like to do with them. Let's start simple by including some custom HTML in an otherwise vanilla Shiny application.

Custom HTML links in Shiny

This application makes use of data downloaded from a website I use a lot in my daily work, Patient Opinion (`www.patientopinion.org.uk/`). Patient Opinion lets users of health services tell their stories, and my organization makes extensive use of it to gather feedback about our services and improve them. This application uses data downloaded from the site and allows users to see the rate at which stories are posted that relate to different parts of the organization. On Patient Opinion, a custom HTML button will take the users straight from the application and onto the search page for that service area.

ui.R

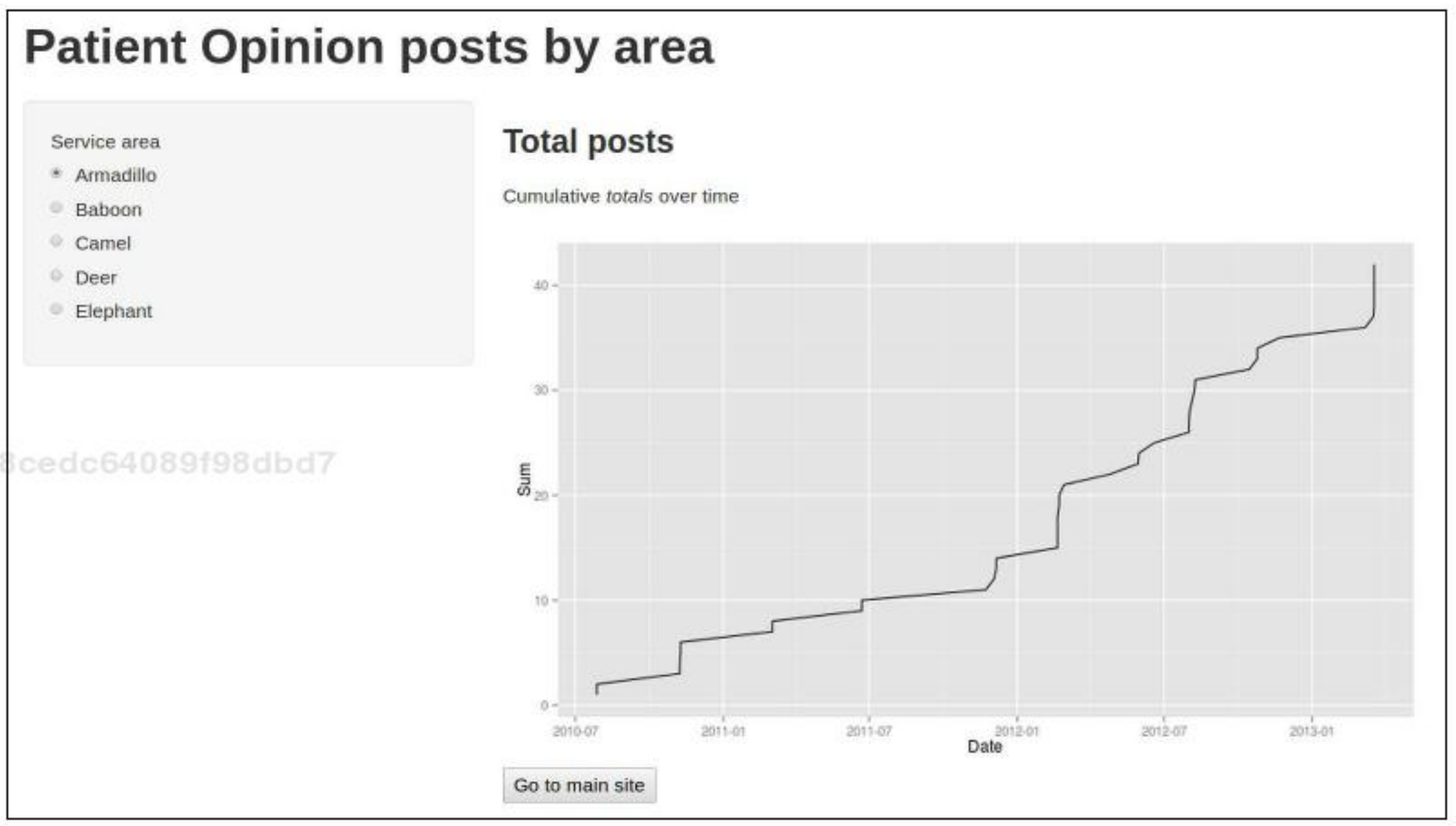
Let's take a look at the `ui.R` first:

```
####################################
### custom HTML output - ui.R ###
####################################

library(shiny)

shinyUI(pageWithSidebar(

  headerPanel("Patient Opinion posts by area"),

  sidebarPanel(

    radioButtons("area", "Service area",
```

```
                c("Armadillo", "Baboon",
                  "Camel", "Deer", "Elephant"),
                selected = "Armadillo")
    ),

    mainPanel(
      h3("Total posts"),
      HTML("<p>Cumulative <em>totals</em> over time</p>"),
      plotOutput("plotDisplay"),
      htmlOutput("outputLink")
      )
  ))
```


Hopefully, you should remember the `h3("...")` function and all the other helper functions from the previous chapter. Just type `?p` at the console for the full list. I have also included the `HTML()` function which marks text strings as HTML, avoiding the HTML escaping, which would otherwise render this on the screen verbatim.

The other new part of this file is the `htmlOutput()` function. This, like the `HTML()` function, prevents HTML escaping and allows you to use your own markup, but this time for text passed from `server.R`. Here's the final interface:



server.R

There are only a couple of new commands from Shiny in this example, so let's sharpen our R skills while we are here. The `server.R` file in this example, unlike many of the others in the book, deliberately does a lot of data management and clean-up at the top, before the reactive code. In this case, of course, I could have cleaned the data first and then loaded the clean data in the example. However, usually you will not have this luxury, either because results are loaded online from an API, or because your users will drop their own spreadsheets into the application folder, or some other reason such as this. So, let's look at a more realistic example and put aside Shiny commands for the moment.

server.R – data preparation

Let's look at the data preparation code first:

```
##########################################
##### custom HTML output - server.R ###
##########################################

library(shiny)
library(ggplot2)

# load the data- keeping strings as strings

PO <- read.csv("PO.csv", stringsAsFactors = FALSE)

# create a new variable to hold the area in and fill with blanks

PO$Area <- NA

# find posts that match service codes and label them
# with the correct names

PO$Area[grep("RHARY", PO$HealthServices, ignore.case=TRUE)] <-
  "Armadillo"

PO$Area[grep("RHAAR", PO$HealthServices, ignore.case=TRUE)] <-
  "Baboon"

PO$Area[grep("715", PO$HealthServices, ignore.case=TRUE)] <-
  "Camel"

PO$Area[grep("710i", PO$HealthServices, ignore.case=TRUE)] <-
```

```
  "Deer"

PO$Area[grep("700", PO$HealthServices, ignore.case=TRUE)] <-
  "Elephant"

# create a postings variable to add together for a
# cumulative sum- give it 1

PO$ToAdd <- 1

# remove all missing values for Area
# (since they will never be shown)

PO <- PO[!is.na(PO$Area),]

# API returns data in reverse chronological order- reverse it

PO <- PO[nrow(PO):1,]

# produce cumulative sum column

PO$Sum <- ave(PO$ToAdd, PO$Area, FUN = cumsum)

# produce a date column from the data column in the spreadsheet

PO$Date <- as.Date(substr(PO$dtSubmitted, 1, 10),
                   format = "%Y-%m-%d")
```


After loading the Shiny package and any other packages that are necessary, a comma delimited spreadsheet (`.csv`) is loaded with the `read.csv()` command. You may often want to use `stringsAsFactors = FALSE`. Factor is a special class in R which is useful for statistical applications. A full discussion on the properties of the factor class is rather outside the scope of this book. For now, it is sufficient to say that if you have any strings in your spreadsheet that you want to treat as strings (for example, extracting characters, coercing other variable types such as date, and so on), do ensure that you import them as strings and not as factors as done in the previous example. If you want to use factors for certain particular variables (particularly for ggplot2 which can require factors for some arguments), you can always coerce them later on. Data preparation proceeds as follows:

- A new variable is created and filled with R's missing data value, `NA`. The missing data value is of great use in a lot of R code; here we are using it so that we can easily discard all the datapoints that fail to match to areas.

- The subset operator `[]` is then used with the `grep()` command (familiar to Unix-like OS users and which returns the positions of a character vector matching a search string). This marks all the rows of the newly created empty variable that matches each service code with a name that is meaningful to the end users.
- A helper variable `ToAdd` is then given a value of `1` for all the rows. This will be used to calculate the cumulative total of posts for each area.
- `PO[!is.na(PO$Area),]` is used to return all the rows of the dataset that do not have missing values for the area variable (that is, failed to match). `!is.na(x)` is a useful function that returns the positions of all the non-missing values of x.
- The API for the website returns the data in reverse chronological order, so it is flipped over using the row indices `nrow(PO):1`, that is, a sequence of integers starting at the number of rows of the data and going down to 1.
- The `ave()` function is used to return the cumulative sum (`cumsum`) for each grouping (`PO$Area`).
- The date string (which has the time appended) is shrunk to the correct size using `substr()` and coerced to R's date class using `as.Date()`. The date class can often trip up newcomers, so it is worth having a good read of `?as.Date()`. Coercing a character string and not a factor, and ensuring that you specify the format of the string properly, should get you over the common pitfalls.

Again, don't worry too much if you don't follow all of the R code. Learning R is really a book in itself. I've included it here to help you get used to the kind of things that you might want to do, and to show you the commonly used shortcuts and pitfalls for beginners. Let's have a look at the `server.R` file.

server.R – server definition

This file produces a plot of the cumulative totals of postings and produces a nicely formatted HTML button ready to post straight into the UI:

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

  output$plotDisplay <- renderPlot({

    # select only the area as selected in the UI

    toPlot = PO[PO$Area == input$area,]

    print(
      ggplot(toPlot, aes(x = Date, y = Sum)) + geom_line()
```

```
  )
})
output$outputLink <- renderText({
  # switch command in R as in many other programming languages
  link <- switch(input$area,
                 "Armadillo" =
    "http://www.patientopinion.org.uk/services/rhary",
                 "Baboon" =
    "http://www.patientopinion.org.uk/services/rhaar",
                 "Camel" =
    "http://www.patientopinion.org.uk/services/rha_715",
                 "Deer" =
    "http://www.patientopinion.org.uk/services/rha_710i",
                 "Elephant" =
    "http://www.patientopinion.org.uk/services/rha_700"
  )

  # paste the HTML together

  paste0('<form action="', link, '"target="_blank">
                 <input type="submit" value="Go to main site">
                 </form>')
  })
})
```


You can see the subsetting again carried out with our old friend `[]` and a `ggplot()` call in the plot function. Just remember to wrap it in `print()` (as done in the previous chapter).

The HTML button is created very easily using the `switch()` command, and `paste0()` which concatenates strings with no spaces. With that, our newly created object `output$ouputLink` is ready to be sent straight to the UI and included as raw HTML.

Minimal HTML interface

Now that we have dipped our toes into HTML, let's build a (nearly) minimal example of an interface entirely in HTML. To use your own HTML in a Shiny application, create the `server.R` file as you normally would. Then, instead of a `ui.R` file, create a folder called `www` and place a file called `index.html` inside this folder. This is where you will define your interface.

index.html

Let's look at each chunk of `index.html` in turn:

```
<!------------------------------>
<!--Minimal example- HTML UI -->
<!------------------------------>

<html>

<head>
  <title>HTML minimal example</title>
  <script src="shared/jquery.js" type="text/javascript"></script>
  <script src="shared/shiny.js" type="text/javascript"></script>
  <link rel="stylesheet" type="text/css" href="shared/shiny.css"/>
  <style type = "text/css">
    body {
      background-color: #ecf1ef;
    }

    #navigation {
      position: absolute;
      width: 300px;
    }

    #centerdoc {
      max-width: 600px;
      margin-left: 350px;
      border-left: 1px solid #c6ec8c;
      padding-left: 20px;
    }
  </style>

</head>
```


The `<head>` section contains some important setup for Shiny, loading the JavaScript and jQuery scripts which make it work, as well as a stylesheet for Shiny. You will need to add some CSS of your own unless you want every element of the interface and output to be displayed as a big list down the screen, and the whole thing to look very ugly. For simplicity, I've added some very basic CSS in the `<head>` section; you could, of course, use a separate CSS file and add a link to it just as `shiny.css` is referenced.

The body of the HTML contains all the input and output elements that you want to use, and any other content that you want on the page. In this case, I've mixed up a Shiny interface with a picture of my cats, because no web page is complete without a picture of a cat! Have a look at the following code:

```
<body>

  <h1>Minimal HTML UI</h1>

<div id = "navigation">

  <p>
    <label>Title for graph:</label><br />
    <textarea name="comment" rows = "4"
    cols = "30">My first graph</textarea>
  </p>

  <p>
    <label>What sort of graph would you like?</label><br />
    <input type="radio" name="graph"
    value="1" title="Straight line" checked>Linear<br>
    <input type="radio" name="graph" value="2"
    title="Curve" >Quadratic<br>
  </p>

    <label>Here's a picture of my cats</label><br />
    <img src="cat.jpg" alt="My cats"
    width="300" height = "300">

</div>

<div id = "centerdoc">

  <div id="textDisplay" class="shiny-text-output"></div>
  <br/ >

  <div id="plotDisplay" class="shiny-plot-output"
  style="width: 80%; height: 400px"></div>

</div>

</body>

</html>
```

There are three main elements: a title and two `<div>` sections, one for the UI and one for the output. The UI is defined within the navigation `<div>`, which is left aligned. Recreating Shiny widgets in HTML is pretty simple and you can also use HTML elements that are not given in Shiny. Instead of replacing the `textInput()` widget with `<input type="text">` (which is equivalent), I have instead used `<textarea>`, which allows more control over the size and shape of the input area.

The `radioButtons()` widget can be recreated with `<input type = "radio">`. You can see that both get a name attribute, which is referenced in the `server.R` file as `input$name` (in this case, `input$comment` and `input$graph`). Another advantage of using your own HTML is you can add tooltips; I have added these to the radio buttons using the `title` attribute.

The output region is set up with two `<div>` tags: one which is named `textDisplay` and picks up `output$textDisplay` as defined in `server.R`; and the other which is named `plotDisplay` and picks up `output$plotDisplay` from the `server.R` file. In your own code, you will need to specify the class as shown in the previous example, as either `shiny-text-output` (for text), `shiny-plot-output` (for plots), or `shiny-html-output` (for tables or anything else that R will output as HTML). You will need to specify the height of plots (in px, cm, and so on) and can optionally specify width either in absolute or relative (%) terms.

Just to demonstrate that you can throw anything in there that you like, there's a picture of my cats underneath the UI. You will, of course, have something a bit more sophisticated in mind. Add more `<div>` sections, links, pictures and just whatever you like.

server.R

Let us have a quick look at the `server.R` file:

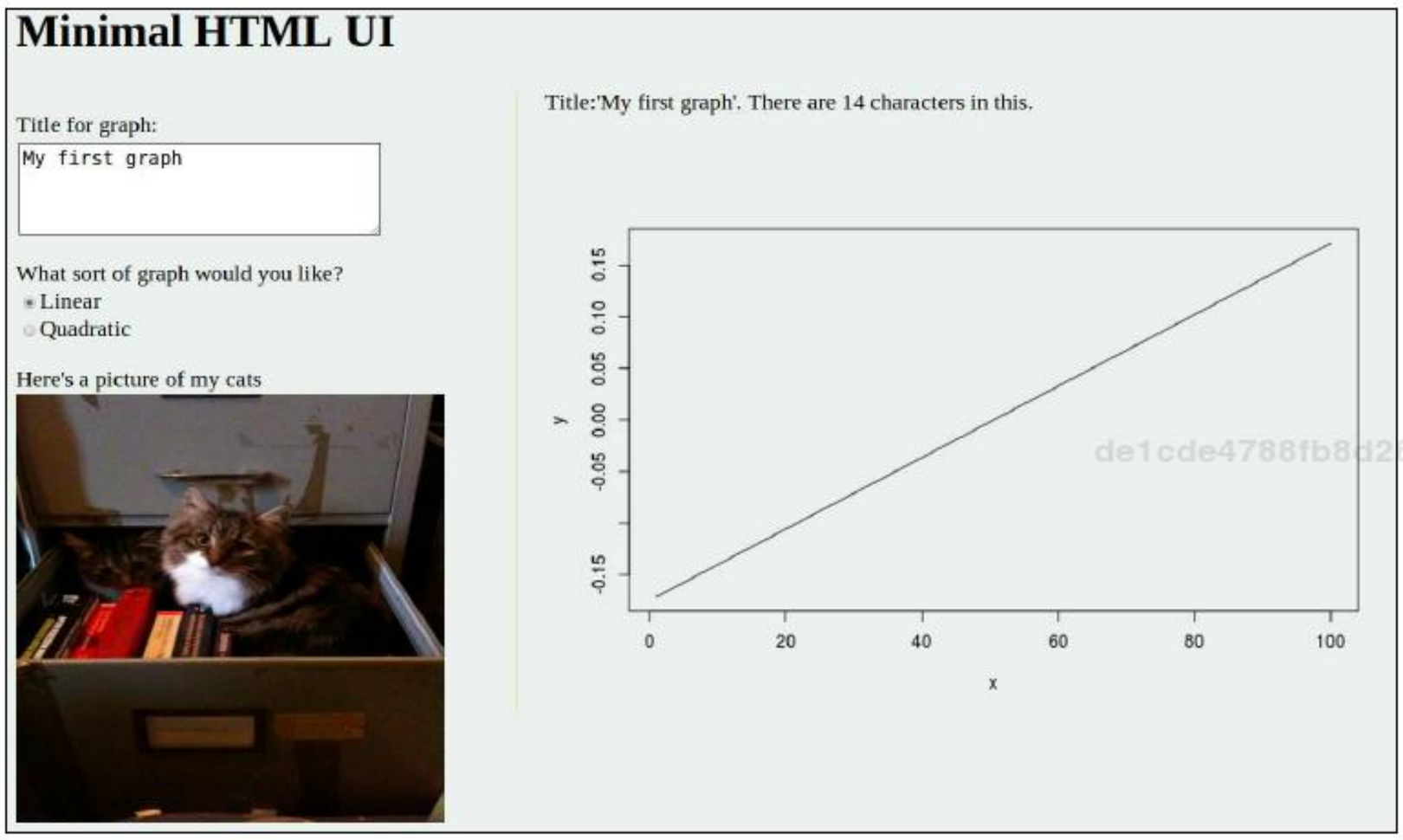
```
##################################################
##### minimal example for HTML- server.R #####
##################################################

library(shiny)

shinyServer(function(input, output) {

  output$textDisplay <- renderText({
    paste0("Title:'", input$comment,
```

```
            "'. There are ", nchar(input$comment),
            " characters in this."
      )
    })

    output$plotDisplay <- renderPlot({

      par(bg = "#ecf1ef") # set the background color

      plot(poly(1:100, as.numeric(input$graph)), type = "l",
        ylab="y", xlab="x")

    })

  })
```


Text handling is done as before. You'll notice that the `renderPlot()` function begins by setting the background color to the same as the page itself (`par(bg = "#ecf1ef")` and for more graphical options in R, see `?par`). You don't have to do this, but the graph's background will be visible as a big white square if you don't. The next chapter will tell you how to draw your graph as a **png** and handle the transparency yourself.

The actual plot itself uses the `poly()` command to produce a set of numbers from a linear or quadratic function according to the user input (that is, `input$graph`). Note the use of `as.numeric()` to coerce the value we get from the radio button definition in `index.html` from a string to a number. This is a common source of error in Shiny code and you must remember to keep track of how variables are stored, whether as lists, strings, or other variable types; and either coerce them into place (as done here), or coerce them all in one go using a reactive function. The latter option can be a good idea to make your code less fiddly and buggy, since it removes the need to keep track of variable types in every single function you write. There is more about defining your own reactive functions and passing data around a Shiny instance in the next chapter. The type `="l"` argument returns a line graph, and the `xlab` and `ylab` arguments give labels to the x and y axes.

The following screenshot shows the finished article:



JavaScript and Shiny

With Shiny, JavaScript, and jQuery, you can build pretty much anything you can think of; moreover, Shiny and jQuery will do a lot of the heavy lifting which means fairly minimal amounts of code will be required. We are going to have a look at another couple of *toy* examples. Firstly, we will look at using JavaScript to manipulate the inputs of a Shiny application, and then at using jQuery to manipulate the outputs. Please note that these examples do not represent the best practice in coding as they do not make the best use of CSS, HTML, or jQuery. They are just there to demonstrate the principles and show you how easy it is. In your own applications, you will need to make use of HTML, JavaScript (and/or jQuery), and CSS in the most appropriate and efficient way.

ui.R

This example also includes more examples related to including custom HTML without writing the whole thing out in HTML yourself. Let's have a look at the `ui.R` file:

```
##################################################
#### Animating text with JavaScript- ui.R #####
##################################################
```

```
library(shiny)

shinyUI(pageWithSidebar(
  headerPanel("Text based animations"),

  sidebarPanel(
    h3("Let's animate something!"),          # heading helper
    p("Please enjoy the
      animation responsibly"),               # paragraph helper
    tags$textarea(id="textArea",             # tags$XX for
                  "Please enter text here"), # generating HTML
    tags$input(type = "button",
               id = "animate",
               value = "Animate!",
               onClick = "buttonClick()")    # reference to JS
  ),

  mainPanel(
    tags$canvas(id="myCanvas", # graphical output area
                width="500",
                height="250"),
    includeHTML("textSend.js"), # include JS file
    textOutput("textDisplay")
  )
))
```


There are two things in this file that you haven't seen before. The first is the `tags$xxx()` function which will generate HTML for you. The `tags$textarea(id="textArea", "Please enter text here")` call generates the following:

```
<textarea id="textArea" class="shiny-bound-input">Please enter text
here</textarea>.
```

Similarly, the whole `tags$input(...)` call generates the following:

```
<input type="button" id="animate" value="Animate!"
onclick="buttonClick()">.
```

The second thing that you haven't seen before is the `includeHTML()` function. This allows you to link to a file that contains a lot of HTML (in this case, a JavaScript definition), rather than cluttering up your `ui.R` with it. You could very well include plain HTML using this function.

server.R

The `server.R` file is unchanged from our original minimal example:

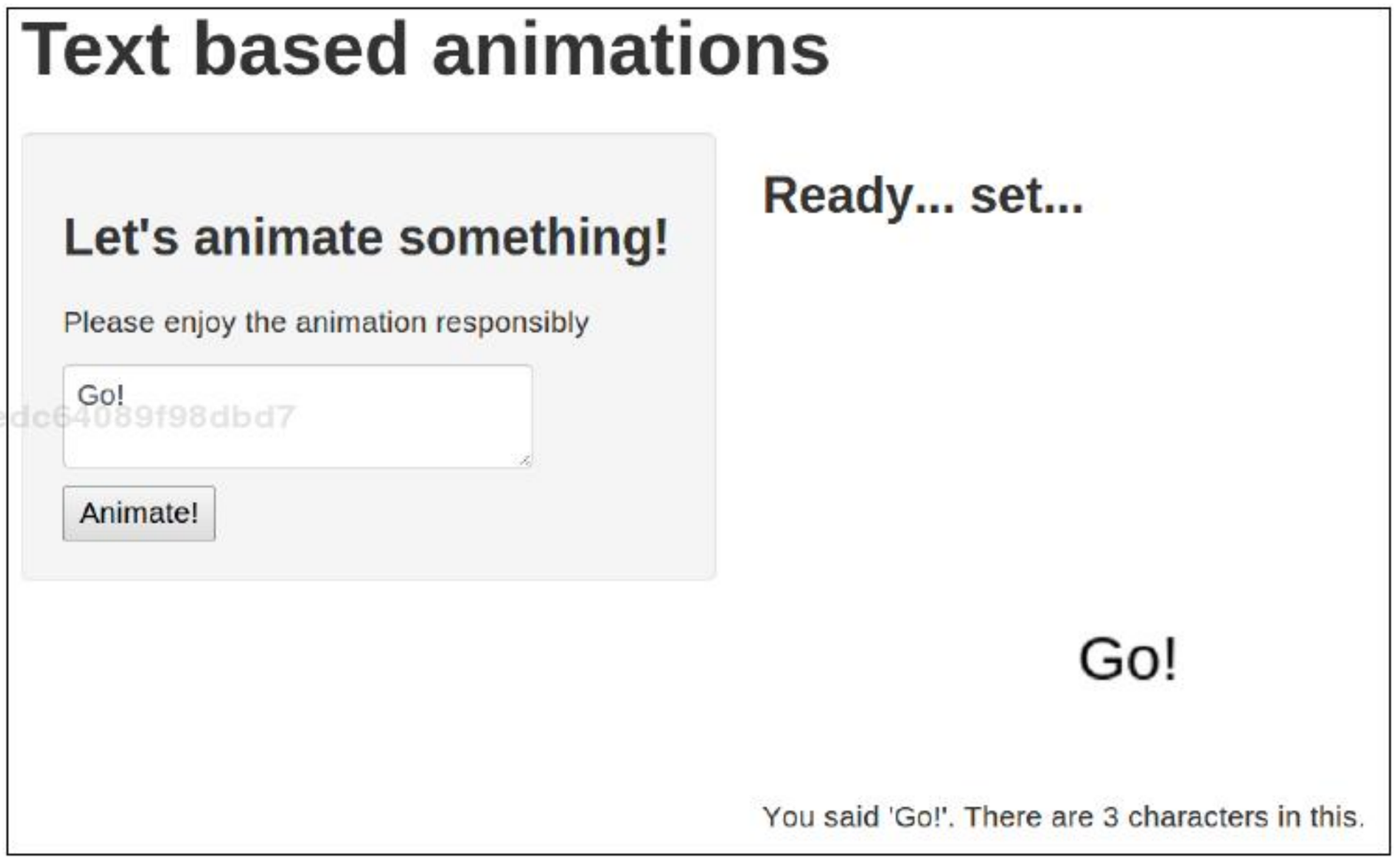
```
##################################################
##### Animating text with JavaScript - server.R ####
##################################################

library(shiny)

shinyServer(function(input, output) {

  output$textDisplay <- renderText({ # handle Shiny
                                     # text function
    paste0("You said '", input$textArea,
           "'. There are ", nchar(input$textArea),
           " characters in this."
    )
  })
})
```

Of course, you can do much more processing than this if you wish. The JavaScript file contains no surprises, and functions just as it does in any other web application:

```
<script type="text/javascript">

function buttonClick(){

  // get and set up the drawing canvas

  var c=document.getElementById("myCanvas");
  var ctx=c.getContext("2d");
  ctx.font="30px Arial";

  // get the text from the UI

  var text = document.getElementById("textArea").value;

  // set up positional variables

  var textX = 150;
  var textY = 1;

  // define move function
```

```
function move(){

  ctx.clearRect(0, 0, c.width, c.height);
  ctx.fillText(text, textX, textY * 5);

  if(textY++ < 40){
    setTimeout(move, 25); // delay between frames
  }
 }

  move(); // call function
}

</script>
```

We won't look in detail at the JavaScript because that is rather out of the scope of this discussion. The important things to note are the first code chunk, which picks up the drawing canvas we drew in the `ui.R` file and sets it up; and the second code chunk, which picks up the input from the `textarea` that we defined in `ui.R`. The rest of the code just draws the text on the screen and then animates it so that it falls down the frame. Here is the screenshot that displays it:



You will, I am sure, wish to produce something a little more sophisticated than this!

jQuery

For the ultimate quick and clean code, let's add some jQuery. We are going to add mouseover row highlighting (that is, coloring in the rows of a table when the mouse pointer is on them) for a table from Shiny (this can be done in CSS, of course, but this is just an example) and allow the user to bold individual cells by clicking on them, as well as producing a pop-up information box about the dataset.

index.html – body

We'll skip the head for now and look at the body of the `index.html` file:


```
<body>
    <h1>jQuery example</h1>

    <div id = "navigation">

      <label for="dataSet">Select dataset</label>
      <select id="dataSet">
        <option value="iris" selected="selected">
          Iris data</option>
        <option value="USPersonalExpenditure">
          Personal expenditure data</option>
        <option value="CO2">CO2 data</option>
      </select>

    </div>

    <div id = "centerdoc">

      <div id="datatext" class="shiny-text-output"></div>

      <div id="hiddentext" style = "text-indent: 100%;
        white-space: nowrap; overflow: hidden"
        class="shiny-text-output">
      </div>

      <div id="dataset" class="shiny-html-output"></div>

    </div>
</body>
```

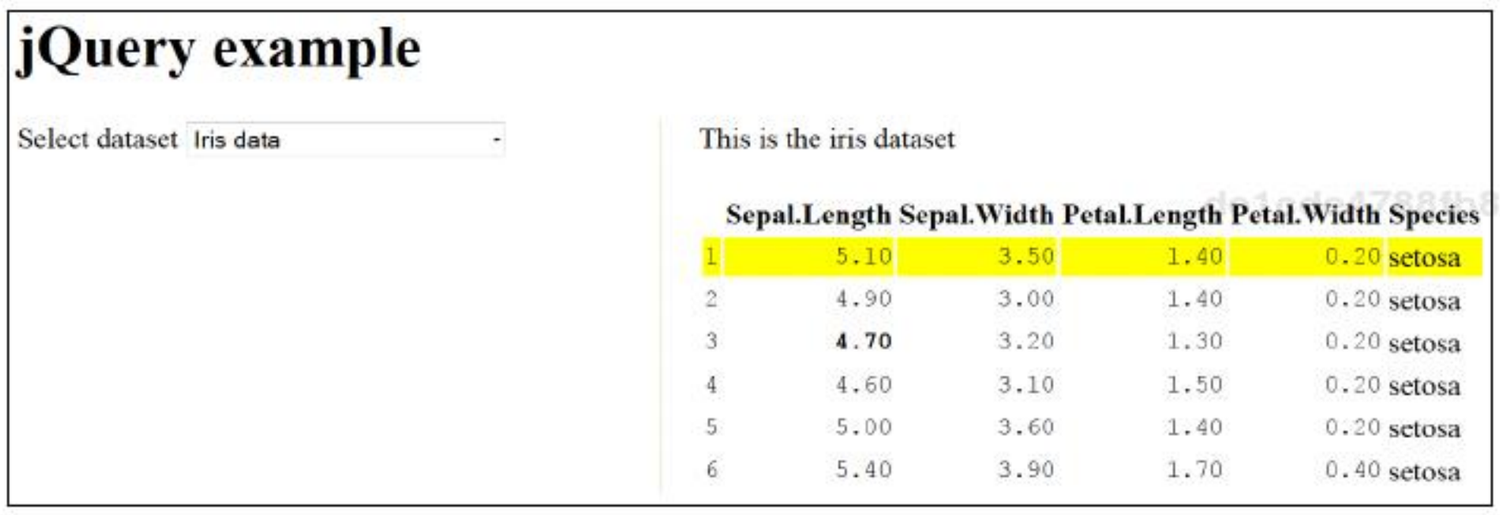
The interface, as you can see, allows users to select one of three datasets which are included in R. There are two outputs that are visible, which are some text followed by a table that will be specified within the `server.R` file (you can see them in the previous code snippet, the `<div>` sections with `id = "datatext"` and `id = "dataset"`). A further `<div>` section (with `id = "hiddentext"`) allows R to generate some text, so make it available to jQuery but without displaying it on the screen until the user requests it. Let's now look at the `server.R` file.

server.R

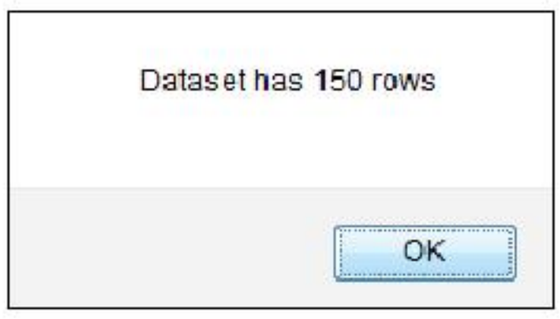
Following is the `server.R` file:


```
library(shiny)

shinyServer(function(input, output) {

  output$dataset <- renderTable({

    theData = switch(input$dataSet,
                     "iris" = iris,
                     "USPersonalExpenditure" =
                        USPersonalExpenditure,
                     "CO2" = CO2)

    head(theData)

  })

  output$datatext <- renderText({

    paste0("This is the ", input$dataSet, " dataset")

  })

  output$hiddentext <- renderText({

    paste0("Dataset has ", nrow(switch(input$dataSet,
                                       "iris" = iris,
                                       "USPersonalExpenditure" =
                                        USPersonalExpenditure,
                                       "CO2" = CO2)), " rows")

  })
})
```

The function within `renderTable()` quite simply takes the string sent from the interface and returns the dataset within R with the same name. It then displays the first few rows of the dataset using `head()`, which is returned as an HTML table. There are two calls made to `renderText()`, as can be seen. The first returns a text string describing which dataset has been selected. The second returns a description of the number of rows in the dataset. This will be hidden from the user and is only accessible via jQuery. Before we go into detail, here is the finished interface:



As you can see in the previous screenshot, the first row is highlighted. This is achieved through a mouseover (which works on any row). The third value of `Sepal.Length` is in bold and this is achieved through a mouse click. Double clicking on the text above the table brings up a message about the dataset, as shown in the following screenshot:



This, of course, is the text that we generated and hid, as we saw in the `server.R` and `index.html` files. Let's look at the jQuery to do this.

jQuery

Like before, you can keep the jQuery code wherever you like: in a text file, verbatim in the `<head>` of your html, or using a call to `includeHTML()` from a `ui.R` file. As usual, wrap your code in the following manner:

```
$(document).ready(function(){
  ...
})
```

Please have a look at the piece of code in standard jQuery:

```
$('tr').mouseover(function(){
    $(this).css('background-color', 'yellow');
});
```

This will not work because your output will be redrawn, and so you will need to access all the elements that will be drawn as well as those that already are. Rewrite the previous code in the following manner (it is a piece of code from Joe Cheng of RStudio):

```
$(document).on("mouseover", "tr", function(evt) {
  $(this).css('background-color', 'yellow');
})
```


The previous is the `mouseover` code that handles row highlighting, and following is the `mouseout` code to put it back to normal once the pointer leaves:

```
$(document).on("mouseout", "tr", function(evt) {
  $(this).css('background-color', 'transparent');
});
```

Applying bold effects to individual cells is achieved through the following code snippet. As you can see, the function starts by clearing bold formatting from all the cells (in case a different cell has already been highlighted by the user) and then bolds the cell that has been clicked:

```
$(document).on("click", "td", function(evt) {
  $('td').css('font-weight', 'normal');
  $(this).css('font-weight', 'bold');
})
```

Lastly, the following code snippet describes a function that listens for a double-click on the text that describes the dataset, and then gives more information about the data, which we have placed on the screen and hidden:


```
$(document).on("dblclick", "#datatext", function(evt) {
  alert($('#hiddentext').text());
})
```

As with the JavaScript example, none of these functions are going to win any prizes for UI design, but they do hopefully illustrate some general things that are very easy to accomplish. Following are some examples of things you might like to try in your own applications:

- Click to expand sets of rows in large tables

- Custom highlighting of table cells within a user-set range (note that this can be done without jQuery, using pure Shiny code, but it is more difficult this way)
- Mouseover help text to provide additional documentation for a Shiny application

Exercise

If you haven't already been tempted, now is definitely a good time to have a go at building your own application with your own data. The next chapter covers advanced topics in Shiny and, though you are welcome to plough on, a little practical experience with the functions will stand you in good stead for the next chapter. If you're interested in sharing your creations right away, feel free to jump to *Chapter 5, Running and Sharing Your Creations*.

How you go about building your first application will very much depend on your previous experience and what you want to achieve with Shiny, but as with everything in life, it is better to start simple. Start with the minimal example given in the previous chapter and put in some data that's relevant to you. Shiny applications can be hard to debug (compared to interactive R sessions, at least), so in your early forays, keep things very simple. For example, instead of drawing a graph, start with a simple `renderText()` call and just print the first few values of a variable. This will, at least, let you know that your data is loading okay and the server and UI are communicating properly. Always make sure that any code you write in R (graphs, tables, data management, and so on) works in a plain interactive session, before you put it into a Shiny application!

Probably the most helpful and simple debugging technique is to use `cat()` to print to the R console. There are two main reasons why you should do this. The first is to put in little messages to yourself, for example, `cat("This branch of code executed")`. The second is to print the properties of R objects if you are having problems relating to data structure, size, or type. `cat(str(x))` is particularly useful and will print a summary of any kind of R object, whether it is a list, a dataframe, a numeric vector, or anything else.

The other useful method is a standard method of debugging in R, `browser()`, which can be put anywhere in your code. As soon as it is executed, it halts the application and enters the debug mode (see `?browser`).

Once you have the application working, you can start to add custom HTML using Shiny's built-in functions or rewrite `ui.R` into `index.html`. The choice here really depends on how much HTML you want to include. Although, in theory, you can create very large HTML interfaces in Shiny using `.html` files referenced by the `includeHTML()` command, you will end up with a rather confusing list of markups scattered across different files. Rewriting to raw HTML is likely to be the easier option in most cases. If you are already proficient in JavaScript and/or jQuery, then you may like to have a go at using them with a Shiny application. If not, you can leave this for now or perhaps just modify the code included in this chapter to see whether you can get different and interesting effects.

Summary

This chapter has put quite a heap of tools in your Shiny toolbox. You have learned how to use custom HTML straight from a minimal `ui.R` UI setup, and how to build the whole thing from scratch using HTML and CSS. You have also looked at some data management and cleaning in R, and at some examples of Shiny applications using JavaScript and jQuery. Hopefully, by now, you should have made your own application, whether in pure Shiny or with your own HTML markup, and perhaps experimented with JavaScript/jQuery. In the next chapter, we are going to learn more about higher control over Shiny applications, including controlling reactivity, scoping and passing variables, and a variety of input/output functions.