

Web Application Development with R Using Shiny

Third Edition

Build stunning graphics and interactive data visualizations to deliver cutting-edge analytics



Packt

www.packt.com

Chris Beeley and Shitalkumar R. Sukhdev

Web Application Development with R Using Shiny
Third Edition

Build stunning graphics and interactive data visualizations to deliver cutting-edge analytics

Chris Beeley
Shitalkumar R. Sukhdev

Packt

BIRMINGHAM - MUMBAI

Web Application Development with R Using Shiny Third Edition

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Acquisition Editor: Devanshi Doshi

Content Development Editor: Aishwarya Gawankar

Technical Editor: Rutuja Vaze

Copy Editor: Safis Editing

Project Coordinator: Sheejal Shah

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Graphics: Alishon Mendonsa

Production Coordinator: Arvindkumar Gupta

First published: October 2013

Second published: January 2016

Third edition: September 2018

Production reference: 1260918

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78899-312-8

www.packtpub.com



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the authors

Chris Beeley has been using R and other open source software for ten years to better capture, analyze, and visualize data in the healthcare sector in the UK. He is the author of *Web Application Development with R Using Shiny*. He works full-time, developing software to store, collate, and present questionnaire data using open technologies (MySQL, PHP, R, and Shiny), with a particular emphasis on using the web and Shiny to produce simple and attractive data summaries. Chris is working hard to increase the use of R and Shiny, both within his own organization and throughout the rest of the healthcare sector, as well to enable his organization to better use a variety of other data science tools. Chris has also delivered talks about Shiny all over the country.

Shitalkumar R. Sukhdeve is a senior data scientist at PT Smartfren Telecom Tbk, Jakarta, Indonesia. On his career journey, he has worked with Reliance Jio as a data scientist, entrepreneur, and corporate trainer. He has trained over 1,000 professionals and students and has delivered over 200 lectures on R and machine learning. Research and development in AI-driven self-optimizing networks, predictive maintenance, optimal network quality, anomaly detection, and customer experience management for 4G LTE networks are all areas of interest to Shitalkumar. He is very experienced with R, Spark, R Shiny, H2O, Python, KNIME, the Hadoop ecosystem, MapReduce, Hive, and configuring the open source R Shiny server for machine learning models and dashboard deployment.

I would like to thank my father, Mr. Rajendra Sukhdeve; mother, Mrs. Manju Sukhdeve; wife, Sandika; family; and friends for always believing in me and allowing me to devote time to writing this book. My special thanks to Mr. Nikola Sucevic, Mr. Bhupesh Daheria, and Mr. Pramod Kolhatkar for giving me an opportunity to work with them and explore data science. Thanks to Packt Publishing and team for their continuous support and guidance.

About the reviewer

Abhinav Agrawal has more than 13 years' IT experience and has worked with top consulting firms and US financial institutions. His expertise lies in the banking and financial services domain and he is a seasoned project/program management professional with a passion for data analytics, machine learning, artificial intelligence, robotics process automation, digital transformation, and emerging digital payments solutions. He started using R and Shiny in 2014 to develop web-based analytics solutions for clients. He works as a program manager and is a freelance R instructor and R Shiny consultant. In his spare time, he loves to mentor data science students, make data analytics-related instructional videos on YouTube, and share knowledge with the community.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Title Page	
Copyright and Credits	
Web Application Development with R Using Shiny Third Edition	
www.PacktPub.com	
Why subscribe?	
Packt.com	
Contributors	
About the authors	
About the reviewer	
Packt is searching for authors like you	
Preface	
Who this book is for	
What this book covers	
To get the most out of this book	
Download the example code files	
Download the color images	
Conventions used	
Get in touch	
Reviews	
1. Beginning R and Shiny	
Installing R	
The R console	
Code editors and IDEs	
Learning R	
Getting help	
Loading data	
Data types and structures	
Dataframes, lists, arrays, and matrices	
Variable types	
Functions	
Objects	
Base graphics and ggplot2	
Bar chart	
Line chart	
Introduction to the tidyverse	
Ceci n'est pas une pipe	
Gapminder	
A simple Shiny-enabled line plot	

Installing Shiny and running the examples

Summary

2. Shiny First Steps

Types of Shiny application

Interactive Shiny documents in RMarkdown

A minimal example of a full Shiny application

The ui.R of the minimal example

A note on HTML helper functions

The finished interface

The server.R of the minimal example

The program structure

An optional exercise

Embedding applications in documents

Widget types

The Gapminder application

The UI

Data processing

Reactive objects

Outputs

Text summary

Trend graphs

A map using leaflet

Advanced layout features

Summary

3. Integrating Shiny with HTML

Running the applications and code

Shiny and HTML

Custom HTML links in Shiny

ui.R

server.R

A minimal HTML interface

index.html

server.R

Including a Shiny app on a web page

HTML templates

Inline template code

server.R

ui.R and template.html

Defining code in the ui.R file

ui.R

Take a step back and rewind

Exercise

Debugging

Bootstrap 3 and Shiny

Summary

4. Mastering Shiny's UI Functions

Shiny's layout functions

Simple

Complete

Do it yourself

Combining layout functions

Streamlining the UI by hiding elements

Naming tabPanel elements

Beautiful tables with DataTable

Reactive user interfaces

The reactive user interface example – server.R

The reactive user interface example – ui.R

Progress bars

Progress bar with shinycssloaders

Modals

Alternative Shiny designs

Summary

5. Easy JavaScript and Custom JavaScript Functions

JavaScript and Shiny

Example 1 – reading and writing the DOM

ui.R

appendText.js

Example 2 – sending messages between client and server

ui.R

server.R

dropdownDepend.js

Shinyjs

Extendshinyjs

ui.R

server.R

JavaScript

Responding to events in JavaScript

htmlwidgets

Dygraphs

rCharts

d3heatmap

threejs

Summary

6. Dashboards

Applications in this chapter

Flexdashboards

- Sidebar application with extra styling
 - Adding icons to your UI
 - Using shinythemes
- Using the grid layout
 - ui.R
- Full dashboard
 - Notifications
 - Info boxes
 - ui.R
 - Google Charts gauge
 - Resizing the Google chart
 - ui.R
- Summary

7. Power Shiny

- Animation
- Reading client information and GET requests in Shiny
- Custom interfaces from GET strings
- Downloading graphics and reports
- Downloadable reports with knitr
- Downloading and uploading data
- Bookmarking
 - Bookmarking state
 - Encoding the state into a URL
 - Single-file application
 - Multiple-file application
 - Bookmarking by saving the state to the server
- Interactive plots
- Interactive tables
 - Row selection
 - Column selection
 - Cell Selection
- Linking interactive widgets
- Shiny gadgets
- Adding a password
- Summary

8. Code Patterns in Shiny Applications

- Reactivity in RShiny
 - A closer look at reactivity
 - Controlling specific input with the isolate() function
 - Running reactive functions over time (execution scheduling)
 - Event-handling using observeEvent and eventReactive
 - Functions and modules
 - Shinytest

Debugging
Handling errors (including validate() and req())

Validate

Handling missing input with req()

Profiling R code

Debounce and throttle

Summary

9. Persistent Storage and Sharing Shiny Applications

Sharing over GitHub

An introduction to Git

Using Git and GitHub within Rstudio

Projects in RStudio (h3)

Sharing applications using Git

Sharing using .zip and .tar

Sharing with the world

Shinyapps.io

Shinyapps.io without RStudio

Shiny server

Running Shiny app on Amazon AWS

Scoping, loading, and reusing data in Shiny applications

Temporary data input/output

Persistent data storage

Database using Dplyr, DBI, and POOL

SQL Injection

Summary

Other Books You May Enjoy

Leave a review - let other readers know what you think

Preface

With this book, you will be able to harness the graphical and statistical power of R and rapidly develop interactive and engaging user interfaces using the superb Shiny package, which makes programming for user interaction simple. R is a highly flexible and powerful tool used for analyzing and visualizing data. Shiny is the perfect companion to R, making it quick and simple to share analysis and graphics from R for users to then interact with and query over the web. Let Shiny do the hard work while you spend your time generating content and styling, rather than writing code to handle user inputs. This book is full of practical examples and shows you how to write cutting-edge interactive content for the web, right from a minimal example all the way to fully styled and extensible applications.

This book includes an introduction to Shiny and R and takes you all the way to advanced functions in Shiny as well as using Shiny in conjunction with HTML, CSS, and JavaScript to produce attractive and highly interactive applications quickly and easily. It also includes a detailed look at other packages available for R, which can be used in conjunction with Shiny to produce dashboards, maps, advanced D3 graphics, and much more.

Who this book is for

This book is for anybody who wants to produce interactive data summaries over the web, whether you want to share them with a few colleagues or the whole world.

What this book covers

[Chapter 1](#), *Beginning R and Shiny*, runs through the basics of statistical graphics, data input, and analysis with R. We also discuss data structures and programming basics in R in order to give you a thorough grounding in R before we look at Shiny.

[Chapter 2](#), *Shiny First Steps*, helps you build your first Shiny application. We begin by simply adding interactive content to a document written in Markdown; and then delve deeper into Shiny, building a very primitive and minimal example; and finally, we'll look at more complex applications and the inputs and outputs necessary to build them.

[Chapter 3](#), *Integrating Shiny with HTML*, covers how Shiny works with existing web content in HTML and CSS. We discuss the Shiny helper functions that allow you to add a custom HTML to a standard Shiny application and how to build a minimal example of a Shiny application in your own raw HTML with Shiny running in the background. We'll also get into the use of HTML templates, which make integrating Shiny with HTML easy.

[Chapter 4](#), *Mastering Shiny's UI Functions*, describes all the different ways that Shiny offers to help you achieve the layout and appearance that you want your application to have. It discusses how to show and hide elements of the interface, as well as how to make the interface react to the state of the application. Producing attractive data tables is discussed, as well as how to give your users messages with progress bars and modals.

[Chapter 5](#), *Easy JavaScript and Custom JavaScript Functions*, covers using JavaScript with Shiny, right from adding simple JavaScript right on the page to enhance a program's appearance or functionality, to sending messages to and from the client's browser using messages to and from JavaScript. The use of the `shinyjs` and `htmlwidgets` packages is also discussed, which further add to your ability to add custom or canned JavaScript to a Shiny application.

[Chapter 6](#), *Dashboards*, includes a couple of different types of Shiny dashboard, and describes how to make attractive Shiny dashboards, using color, icons, and a

wide range of inputs and outputs, as well as how to lay them out using the very flexible layout functions, which can be accessed with a Shiny dashboard.

[Chapter 7](#), *Power Shiny*, includes many powerful features of Shiny, such as animating plots, reading client information, and `GET` requests in Shiny. We will go through graphics and report generation and how to download them using `knitr`. Downloading and uploading is also an interesting part of any application, and we'll take a look at it in Shiny with some examples. Bookmarking the state of the application is an add-on to regenerate the output on the application. We will see a demonstration of fast application development using widgets and gadgets. At the end of the chapter, we will see how to authenticate the application using a password.

[Chapter 8](#), *Code Patterns in Shiny Applications*, covers the coding patterns available in Shiny. We will discuss reactivity in R Shiny, controlling specific input with the `isolate()` function, running reactive functions over time, event handling using the `observeEvent` functions and the `shinytest` modules, debugging, handling errors (including `validate()` and `req()`), profiling R code, debounce, and throttle.

[Chapter 9](#), *Persistent Storage and Sharing Shiny Applications*, will explore how to keep your code on GitHub. This chapter will include an introduction to GitHub and how to integrate Git with RStudio. We will also learn how to share your reports and a live application with `shinyapps.io`. This chapter will also focus on the deployment options available, such as Shiny Server and running Shiny in AWS. We will go through some of the concepts that are vital for developing a good Shiny application, such as scoping, loading, and reusing data in Shiny applications. We'll also look at temporary data input/output, permanent data functions, databases, SQL injection, and databases with the `pool` package.

To get the most out of this book

No previous experience with R, Shiny, HTML, or CSS is required to use this book, although you should possess some previous experience with programming in a different language. This book can be used with the Windows, macOS, or Linux operating systems. It requires the installation of R as well as several user-contributed packages within R. R and its associated packages are all available for free. The RStudio IDE is recommended because it simplifies some of the tasks covered in this book, but is not essential. Again, this software is available free of charge.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packt.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Web-Application-Development-with-R-Using-Shiny-third-edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://www.packtpub.com/sites/default/files/downloads/9781788993128_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

`codeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The `[1]` phrase tells you that R returned one result; in this case, 4."

A block of code is set as follows:

```
<ul>
  <li>First bullet</li>
  <li>Second bullet</li>
  <li>Third bullet</li>
</ul>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
tabsetPanel(id = "theTabs",
  tabPanel("Summary", textOutput("summary"),
    value = "summary"),
  tabPanel("Trend", plotOutput("trend"),
    value = "trend"),
  tabPanel("Map", leafletOutput("map"),
    p("Map data is from the most recent year in the selected range"),
    value = "map")
)
```

Any command-line input or output is written as follows:

```
> 2 + 2
[1] 4
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "To set up a new project, go to File | New Project in RStudio."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packt.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

Beginning R and Shiny

R is free and open source, and is the pre-eminent tool for statisticians and data scientists. It has more than 6,000 user-contributed packages, which help users working in fields as diverse as chemistry, biology, physics, finance, psychology, and medical science. R's extremely powerful and flexible statistical graphics greatly help these users in their work.

In recent years, R has become more and more popular, and there are an increasing number of packages for R that make cleaning, analyzing, and presenting data on the web easy for everybody. The Shiny package in particular makes it incredibly easy to deliver interactive data summaries and queries to end users through any modern web browser. You're reading this book because you want to use these powerful and flexible tools for your own content.

This book will show you how, right from when you just start with R, you can build your own interfaces with Shiny and integrate them with your own websites. In this chapter, we're going to cover the following topics:

- Downloading and installing R
- Choosing a code-editing environment/IDE
- Looking at the power of R
- Learning about how RStudio and contributed packages can make writing code, managing projects, and working with data easier
- Installing Shiny and running the examples
- How to use some of Shiny's awesome applications, and some of the elements of the Shiny application that we will build over the course of this book

R is a big subject, and this is a whistle-stop tour, so if you get a little lost along the way, don't worry. This chapter is really all about showing you what's out there, and will both encourage you to delve deeper into the bits that interest you and show you places you can go for help if you want to learn more on a particular subject.

Installing R

R is available for Windows, Mac OS X, and Linux at cran.r-project.org. The source code is also available at the same address. It is also included in many Linux package management systems; Linux users are advised to check before downloading from the web. Details on installing from source or binary for Windows, Mac OS X, and Linux are all available at cran.r-project.org/doc/manuals/R-admin.html.

The R console

Windows and Mac OS X users can run the R application to launch the R console. Linux and Mac OS X users can also run the R console straight from the Terminal by typing `R`.

In either case, the R console itself will look something like the following screenshot:

```
R version 3.4.3 (2017-11-30) -- "Kite-Eating Tree"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> [1]
```

R will respond to your commands right from the Terminal. Let's have a go. Run the following command in the R console:

```
|> 2 + 2
|[1] 4
```

The `[1]` phrase tells you that R returned one result, in this case, `4`. The following command shows you how to print `Hello world`:

```
|> print("Hello world!")
|[1] "Hello world!"
```

The following command shows the multiples of π :

```
|> 1:10 * pi
|[1] 3.141593 6.283185 9.424778 12.566371 15.707963 18.849556
|[7] 21.991149 25.132741 28.274334 31.415927
```

This example illustrates vector-based programming in R. The `1:10` phrase generates the numbers `1:10` as a vector, and each is then multiplied by π , which returns another vector, the elements each being π times larger than the original. Operating on vectors is an important part of writing simple and efficient R code. As you can see, R again indexes the values it returns at the console, with the seventh value being `21.99`.

One of the big strengths of using R is the graphics capability, which is excellent, even in a vanilla installation of R (these graphics are referred to as the base graphics because they ship with R). When adding packages such as `ggplot2` and some of the JavaScript-based packages, R becomes a graphical tour de force, whether producing statistical, mathematical, or topographical figures, or indeed any other type of graphical output. To get a flavor of the power of the base graphics, simply type the following in the Console and see the types of plots that can be made using R:

```
|> demo(graphics)
```

You can also type the following command:

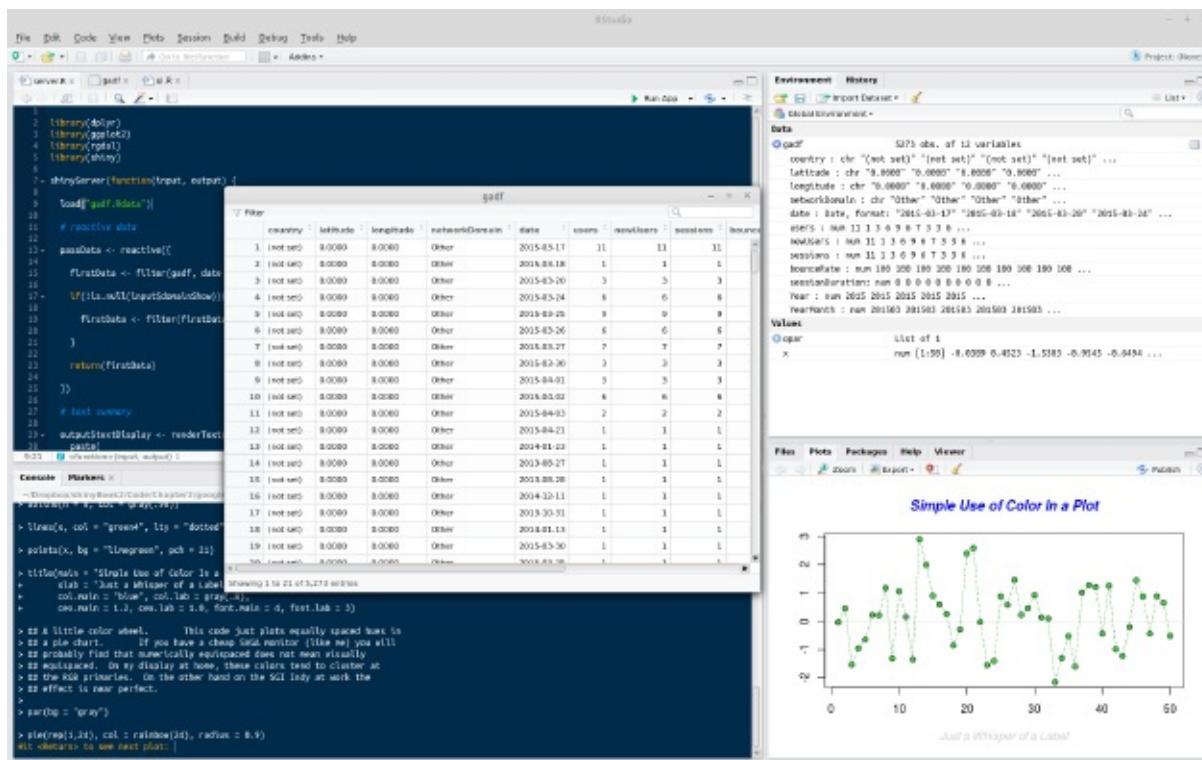
```
|> demo(persp)
```

There will be more on `ggplot2` and base graphics later in the chapter.

Enjoy! There are many more examples of R graphics at r-graph-gallery.com.

Code editors and IDEs

The Windows and OS X versions of R both come with built-in code editors, which allow code to be edited, saved, and sent to the R console. It's hard to recommend that you use this because it is rather primitive. Most users would be best served by RStudio (found at rstudio.com/), which includes project management and version control (including support for Git, which is covered in Chapter 9, *Persistent Storage and Sharing Shiny Applications*), the viewing of data and graphics, code completion, package management, and many other features. The following is an illustrative screenshot of an RStudio session:



As can be seen, in the top-left corner, there is the code-editing pane (with syntax highlighting). Moving clockwise from there will take you to the environment pane (in which you can see the different objects that are loaded into the session), which is the viewing pane containing various options such as Files, Plots, Build, Help, and finally, at the bottom left, the Console. In the middle, there is one of the most useful features of RStudio, the ability to view dataframes. This view can be created by clicking a dataframe in the Environment panel at the top right.

This function also enables sorting and filtering by column.

However, if you already use an IDE for other types of code, it is quite likely that R can be well integrated into it. Examples of IDEs with good R integration include the following:

- Emacs with the Emacs Speaks Statistics plugin
- Vim with the Vim-R plugin
- Eclipse with the StatET plugin

Learning R

There are almost as many uses for R as there are people using it. It is not possible that your specific needs will be covered in this book. However, you probably want to use R to process, query, and visualize data, such as sales figures, satisfaction surveys, concurrent users, sporting results, or whatever types of data your organization processes. For now, let's just take a look at the basics.

Getting help

There are many books and online materials that cover all aspects of R. The name *R* can make it difficult to come up with useful web search hits (substituting CRAN for R can sometimes help); nonetheless, searching for *R tutorial* brings up useful results. Some useful resources include the following:

- An excellent introduction to syntax and data structures in R (at goo.gl/M0RQ5z)
- Videos on using R from Google (at goo.gl/A3uRsh)
- Swirl (at swirlstats.com)
- Quick-R (at statmethods.net)

At the R console, the code phrase `?functionname` can be used to show the help file for a function. For example, `?help` brings up help materials, and using `??help` will bring up a list of potentially relevant functions from installed packages.

Subscribing to and asking questions on the R-help mailing list at stat.ethz.ch/mailman/listinfo/r-help allows you to communicate with some of the leading figures in the R community, as well as many other talented enthusiasts. Read the posting guide and do your research before you ask any questions, because it's a busy and sometimes unforgiving list.

There are two Stack Exchange communities that can provide further help at stats.stackexchange.com/ (for questions about statistics and visualization with R) and [stacoverflow.com/](http://stackoverflow.com/) (for questions about programming with R).

There are many ways to learn R and related subjects online; RStudio has a very useful list on their website at goo.gl/8tx7FP.

Loading data

The simplest way of loading data into R is probably using a comma-separated value (.csv) spreadsheet file, which can be downloaded from many data sources and loaded and saved in all spreadsheet software (such as Excel or LibreOffice). The `read.table()` command imports data of this type by specifying the separator as a comma, or using `read.csv()`, a function specifically for .csv files, as shown in the following command:

```
|> analyticsData = read.table("~/example.csv", sep = ",")
```

Otherwise, you can use the following command:

```
|> analyticsData = read.csv("~/example.csv")
```

Note that unlike other languages, R uses `<-` for assignment as well as `=`. Assignment can be made the other way using `->`. The result of this is that `y` can be told to hold the value of `4` in a `y <- 4` or `4 -> y` format. There are some other, more advanced things that can be done with assignment in R, but don't worry about them now. In this book, I will prefer the `=` operator, since I use this in my own code. Just be aware of both methods so that you can understand the code you come across in forums and blog posts.

Either of the preceding code examples will assign the contents of the `example.csv` file to a dataframe named `analyticsData`, with the first row of the spreadsheet providing the variable names. A dataframe is a special type of object in R, which is designed to be useful for the storage and analysis of data.

RStudio will even take care of loading .csv files for you, if you click on them in the file selector pane (in the bottom right by default) and select Import dataset.... This can be useful to help you get started, but as you get more confident it's really better to do everything with code rather than pointing and clicking. RStudio will, to its great credit, show you the code that makes your pointing and clicking work, so take a note of it and use it to load the data the next time yourself.

Data types and structures

There are many data types and structures of data within R. The following topics summarize some of the main types and structures that you will use when building Shiny applications.

Dataframes, lists, arrays, and matrices

Dataframes have several important features that make them useful for data analysis:

- Rectangular data structures, with the typical use being cases (for example, the days in one month) listed down the rows and variables (page views, unique visitors, or referrers) listed along the columns
- A mix of data types is supported. A typical data frame might include variables containing dates, numbers (integers or floats), and text
- With subsetting and variable extraction, R provides a lot of built-in functionality to select rows and variables within a dataframe
- Many functions include a data argument, which makes it very simple to pass dataframes into functions and process only the variables and cases that are relevant, which makes for cleaner and simpler code

We can inspect the first few rows of the dataframe using the `head(analyticsData)` command. The following screenshot shows the output of this command:

```
> head(analyticsData)
  Day pageViews uniqueVisitors visitDuration
1 2013-06-01      572              21     7.843611
2 2013-06-02      955              36     8.555000
3 2013-06-03      993              48    17.959722
4 2013-06-04      553              41    20.997500
5 2013-06-05      654              16   12.221111
6 2013-06-06      878              47     8.250278
> |
```

As you can see, there are four variables within the dataframe: one contains dates, two contain integer variables, and one contains a numeric variable.

Variables can be extracted from dataframes very simply using the \$ operator, as follows:

```
|> analyticsData$pageViews  
[1] 836 676 940 689 647 899 934 718 776 570 651 816  
[13] 731 604 627 946 634 990 994 599 657 642 894 983  
[25] 646 540 756 989 965 821
```

Variables can also be extracted from dataframes using [], as shown in the following command:

```
|> analyticsData[, "pageViews"]
```

Note the use of a comma with nothing before it to indicate that all rows are required. In general, dataframes can be accessed using `dataObject[x,y]`, with `x` being the number(s) or name(s) of the rows required and `y` being the number(s) or name(s) of the columns required. For example, if the first 10 rows were required from the `pageViews` column, it could be achieved like this:

```
|> analyticsData[1:10, "pageViews"]  
[1] 836 676 940 689 647 899 934 718 776 570
```

Leaving the space before the comma blank returns all rows, and leaving the space after the comma blank returns all variables. For example, the following command returns the first three rows of all variables:

```
|> analyticsData[1:3, ]
```

The following screenshot shows the output of this command:

```
> analyticsData[1:3, ]  
      Day pageViews uniqueVisitors visitDuration  
1 2013-06-01        572             21       7.843611  
2 2013-06-02        955             36       8.555000  
3 2013-06-03        993             48      17.959722  
> |
```

Dataframes are a special type of list. Lists can hold many different types of data, including lists. As with many data types in R, their elements can be named, which can be useful to write code that is easy to understand. Let's make a list of the options for dinner, with drink quantities expressed in milliliters.

In the following example, please also note the use of the `c()` function, which is used to produce vectors and lists by giving their elements separated by commas. R will pick an appropriate class for the return value, string for vectors that contain strings, numeric for those that only contain numbers, logical for Boolean values, and so on:

```
|> dinnerList <- list("Vegetables" =  
|  c("Potatoes", "Cabbage", "Carrots"),  
|  "Dessert" = c("Ice cream", "Apple pie"),  
|  "Drinks" = c(250, 330, 500)  
)
```



Note that code is indented throughout, although entering code directly into the console will not produce indentations; it is done for readability.

Indexing is similar to dataframes (which are, after all, just a special instance of a list). They can be indexed by number, as shown in the following command:

```
|> dinnerList[1:2]  
$Vegetables  
[1] "Potatoes" "Cabbage" "Carrots"  
  
$Dessert  
[1] "Ice cream" "Apple pie"
```

This returns a list. Returning an object of the appropriate class is achieved using `[[[]]]`:

```
|> dinnerList[[3]]  
[1] 250 330 500
```

In this case, a numeric vector is returned. They can also be indexed by name, as shown in the following code:

```
|> dinnerList["Drinks"]  
$Drinks  
[1] 250 330 500
```

Note that this also returns a list.

Matrices and arrays, which, unlike dataframes, only hold one type of data, also make use of square brackets for indexing, with `analyticsMatrix[, 3:6]` returning all rows of the third to sixth columns, `analyticsMatrix[1, 3]` returning just the first row of the third column, and `analyticsArray[1, 2,]` returning the first row of the second column across all of the elements within the third dimension.

Variable types

R is a dynamically typed language, and you are not required to declare the type of your variables when using it. Of course, it is worth knowing about the different types of variable that you might read or write using R. The different types of variable can be stored in a variety of structures, such as vectors, matrices, and dataframes, although some restrictions apply as mentioned previously (for example, matrices must contain only one variable type). The following bullet list contains the specifics of using these variable types:

- Declaring a variable with at least one string in it will produce a vector of strings (in R, the character data type), as shown in the following code:

```
| > c("First", "Third", 4, "Second")
| [1] "First" "Third" "4" "Second"
```

- You will notice that the numeral 4 is converted to a string, "4". This is as a result of coercion, in which elements of a data structure are converted to other data types in order to fit within the types that are allowed within the data structure. Coercion occurs automatically, as in this case, or with an explicit call to the `as()` function—for example, `as.numeric()`, or `as.Date()`.
- Declaring a variable that contains only numbers will produce a numeric vector, as shown in the following code:

```
| > c(15, 10, 20, 11, 0.4, -4)
| [1] 15.0 10.0 20.0 11.0 0.4 -4.0
```

- R, of course, also includes a logical data type, as shown in the following code:

```
| > c(TRUE, FALSE, TRUE, TRUE, FALSE)
| [1] TRUE FALSE TRUE TRUE FALSE
```

- A data type exists for dates, which are often a source of problems for beginners, as shown in the following code:

```
| > as.Date(c("2013/10/24", "2012/12/05", "2011/09/02"))
| [1] "2013-10-24" "2012-12-05" "2011-09-02"
```

- The use of the `factor` data type tells R all of the possible values of a categorical variable, such as gender or species, as shown in the following code:

```
> factor(c("Male", "Female", "Female", "Male", "Male"),
  levels = c("Female", "Male"))
[1] Male   Female Female Male   Male
Levels: Female Male
```

Functions

As you grow in confidence with R, you will want to begin writing your own functions. This is achieved very simply, and in a manner quite similar to many other languages. You will no doubt want to read more about writing functions in R in more detail, but just to give you an idea, the following code is a function called the `sumMultiply` function that adds together `x` and `y` and multiplies the result by `z`:

```
sumMultiply <- function(x, y, z){  
  final = (x+y) * z  
  return(final)  
}
```

This function can now be called using `sumMultiply(2, 3, 6)`, which will return 2 plus 3 times 6, which gives 30.

Objects

There are many special object types within R that are designed to make it easier to analyze data. Functions in R can be polymorphic—that is to say, they can respond to different data types in different ways in order to produce the output that the user desires. For example, the `plot()` function in R responds to a wide variety of data types and objects, including single-dimension vectors (each value of y plotted sequentially) and two-dimensional matrices (producing a scatterplot), as well as specialized statistical objects, such as regression models and time series data. In the latter case, plots that are specialized for these purposes are produced.

As with the rest of this introduction, don't worry if you haven't written functions before, or don't understand object concepts and aren't sure what this all means. You can produce great applications without understanding all these things, but as you do more and more with R, you will start to want to learn more details about how R works and how experts produce R code. This introduction is designed to give you a jumping-off point to learn more about how to get the best out of R (and Shiny).

Base graphics and ggplot2

There are lots of user-contributed graphics packages in R that can produce some wonderful graphics. You may wish to take a look for yourself at the CRAN task view at cran.r-project.org/web/views/Graphics.html. We will have a very quick look at two approaches: base graphics, so called because they form the default graphical environment within a vanilla installation of R, and `ggplot2`, a highly popular user-contributed package produced by Hadley Wickham, which is a little trickier to master than base graphics, but can very rapidly produce a wide range of graphical data summaries. We will cover two graphs that are familiar to everyone: the bar chart and the line chart.

Bar chart

Useful when comparing quantities across categories, bar charts are very simple in base graphics, particularly when combined with the `table()` command. We will use the `mpg` dataset, which comes with the `ggplot2` package; it summarizes the different characteristics of a range of cars. First, let's install the `ggplot2` package. You can do this straight from the console using the following code:

```
|> install.packages("ggplot2")
```

Alternatively, you can use the built-in package functions in IDEs such as RStudio or RKWard. We'll need to load the package at the beginning of each session in which we want to use this dataset, or in the `ggplot2` package itself. From the console, type the following command:

```
|> library(ggplot2)
```

We will use the `table()` command to count the number of each type of car featured in the dataset, as shown in the following code:

```
|> table(mpg$class)
```

This returns a `table` object (another special object type within R) that contains the columns shown in the following screenshot:

```
> table(mpg$class)

 2seater    compact    midsize    minivan    pickup    subcompact      suv
      5         47        41        11        33         35        62
>
```

Producing a bar chart of this object is achieved simply using the following code:

```
|> barplot(table(mpg$class), main = "Base graphics")
```

The `barplot` function takes a vector of frequencies. Where they are named, as is the case in our example (the `table()` command returning named frequencies in

table form), names are automatically included on the *x* axis. The defaults for this graph are rather plain. Explore `?barplot` and `?par` to learn more about fine-tuning your graphics.

We've already loaded the `ggplot2` package in order to use the `mpg` dataset, but if you have shut down R in between these two examples, you will need to reload it by using the following command:

```
| > library(ggplot2)
```

The same graph is produced in `ggplot2` as follows:

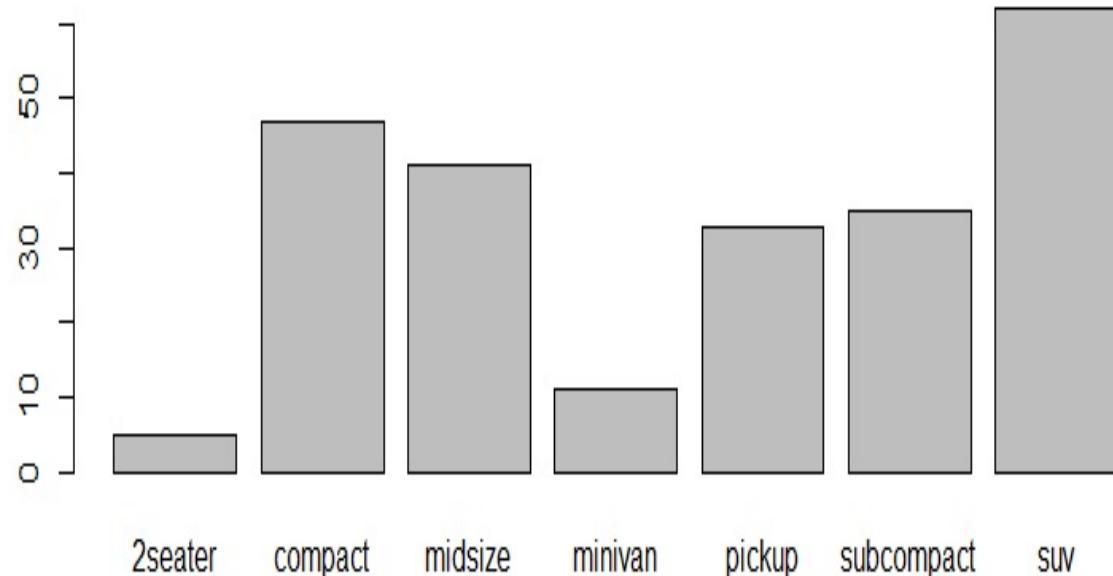
```
| > ggplot(data = mpg, aes(x = class)) + geom_bar() +  
|   ggttitle("ggplot2")
```

This `ggplot` call shows the three fundamental elements of `ggplot` calls: the use of a dataframe (`data = mpg`); the setup of aesthetics (`aes(x = class)`), which determines how variables are mapped onto axes, colors, and other visual features; and the use of `+ geom_xxx()`. A `ggplot` call sets up the data and aesthetics, but does not plot anything. Functions such as `geom_bar()` (there are many others; see `??geom`) tell `ggplot` what type of graph to plot, as well as instructing it to take optional arguments—for example, `geom_bar()` optionally takes a position argument, which defines whether the bars should be stacked, offset, or stretched to a common height to show proportions instead of frequencies.

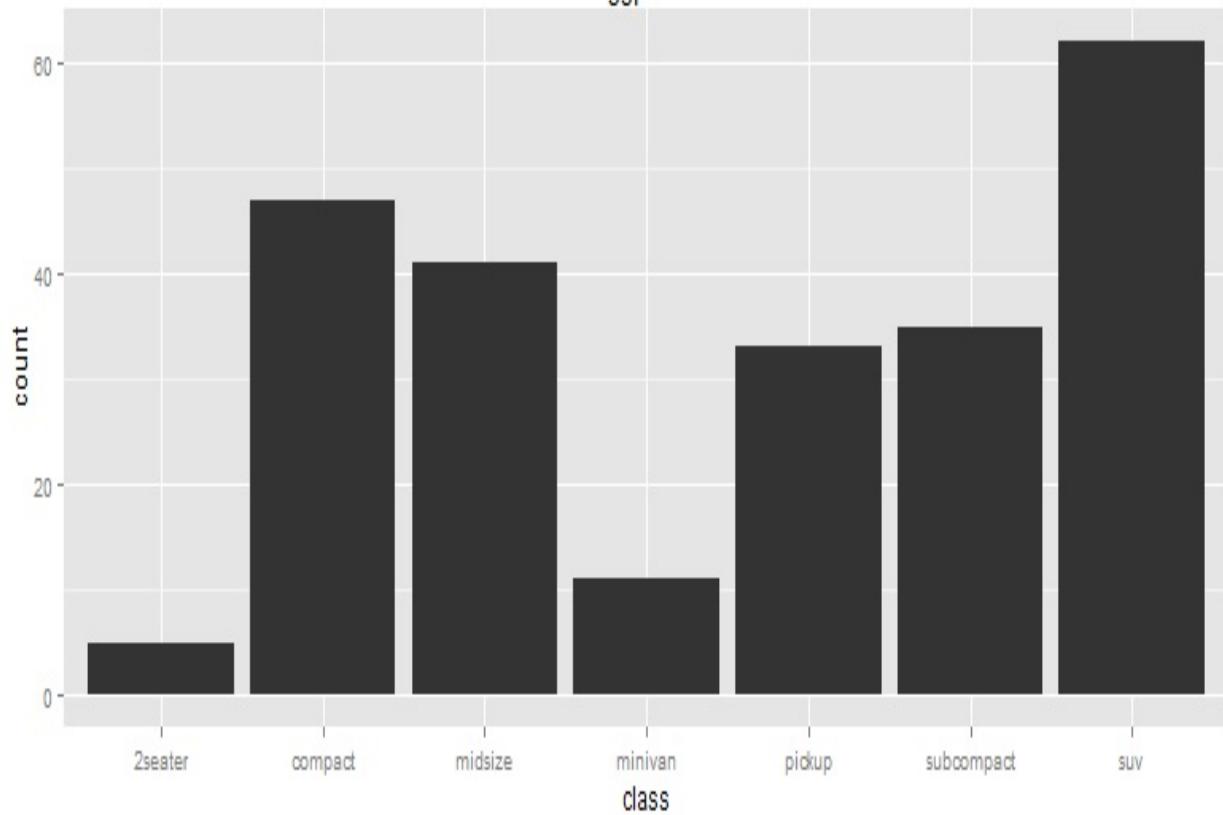
These elements are the key to the power and flexibility that `ggplot2` offers. Once the data structure is defined, ways of visualizing that data structure can be added and taken away easily, not only in terms of the type of graphic (bar, line, or scatter graph) but also the scales and coordinates system (log10, polar coordinates, and so on) and statistical transformations (smoothing data, summarizing over spatial coordinates, and so on). The appearance of plots can be easily changed with preset and user-defined themes, and multiple plots can be added in layers (that is, adding them to one plot) or facets (that is, drawing multiple plots with one function call).

The base graphics and `ggplot` versions of the bar chart are shown in the following screenshot for the purposes of comparison:

Base graphics



ggplot2



Line chart

Line charts are most often used to indicate change, particularly over time. This time, we will use the `longley` dataset, featuring economic variables from between 1947 and 1962, as shown in the following code:

```
| > plot(x = 1947 : 1962, y = longley$GNP, type = "l",
|       xlab = "Year", main = "Base graphics")
```

The `x` axis is given very simply by the `1947 : 1962` phrase, which enumerates all the numbers between 1947 and 1962, and the `type = "l"` argument specifies the plotting of the lines, as opposed to points or both.

The `ggplot` call looks a lot like the bar chart, except with an `x` and `y` dimension in the aesthetics this time. The command looks as follows:

```
| > ggplot(longley, aes(x = 1947 : 1962, y = GNP)) + geom_line() +
|       xlab("Year") + ggtitle("ggplot2")
```

Introduction to the tidyverse

The `tidyverse` is, according to its homepage, "*an opinionated collection of R packages designed for data science*" (see <https://www.tidyverse.org/>). All of the packages are installed using `install.packages("tidyverse")`, and calling `library(tidyverse)` loads a subset of these packages, those considered to have the most value in day-to-day data science. Calling `library(tidyverse)` loads the following packages:

- `ggplot2`: For plotting
- `dplyr`: For data wrangling
- `tidyr`: For tidying (and untidying!) data
- `readr`: Better functions for reading comma- and tab-delimited data and other types of flat files
- `purrr`: For iterating
- `tibble`: Better dataframes
- `stringr`: For dealing with strings
- `forcats`: Better handling of *factors*, an R property used to describe the categories of a variable, described briefly earlier in the chapter

Installing the `tidyverse` also installs the following packages, which then need to be loaded separately with their own `library()` instruction: `readxl`, `haven`, `jsonlite`, `xml2`, `httr`, `rvest`, `DBI`, `lubridate`, `hms`, `blob`, `rlang`, `magrittr`, `glue`, and `broom`. For more details on these packages, consult the documentation at tidyverse.org/.

The key word in this description is *opinionated*. The `tidyverse` is a set of R packages that work with tidy data, either producing it, or consuming it, or both. Tidy data was described by Hadley Wickham in the *Journal of Statistical Software*, Vol 59, Issue 10 (<https://www.jstatsoft.org/article/view/v059i10/v59i10.pdf>). Tidy data obeys three principles:

- Each variable forms a column
- Each observation forms a row
- Each type of observational unit forms a table

For many R users, their first introduction to tidy data will have been `ggplot2`,

which consumes tidy data and requires other types of data to be munged into this form. In order to explain what this means, we will look at a simple example. For the purposes of this discussion, we will ignore the last principle, which is more about organizing groups of datasets rather than individual datasets.

Let's have a look at a simple example of a messy dataset. In the real world, you will find datasets that are a lot messier than this, but this will serve to illustrate the principles we are using here. The following are the first three rows of the medal table for the Pyeongchang Winter Olympics, which took place in 2018, as an R dataframe:

```
| medals = data.frame(country = c("Norway", "Germany", "Canada"),
|   gold = c(14, 14, 11),
|   silver = c(14, 10, 8),
|   bronze = c(11, 7, 10)
| )
```

If we print it at the console, it looks like the following screenshot:

```
> medals
  country gold silver bronze
1 Norway    14     14      11
2 Germany    14      10       7
3 Canada     11      8      10
```

This is perhaps the most common sort of messy data you will come across: great as a summary, certainly intelligible to people watching the Winter Olympics, but not tidy. There are medals in three different columns. A `tidy` dataset would contain only one column for the medal tallies. Let's tidy it up using the `tidyverse` package, which is loaded with `library(tidyverse)`, or you can load it separately with `library(tidyr)`. We can tidy the data very simply using the `gather()` function. The `gather()` function takes a dataframe as an argument, along with key and value column names (which you can set to what you like), and the column names that you wish to be gathered (all other columns will be duplicated as appropriate). In this case, we want to gather everything except the country, so we can use `-variableName` to indicate that we wish to gather everything except `variableName`. The final code looks like the following:

```
| library(tidyr)
| gather(medals, key = Type, value = Medals, -country)
```

This has a nice tidy output, as shown in the following screenshot:

	country	Type	Medals
1	Norway	gold	14
2	Germany	gold	14
3	Canada	gold	11
4	Norway	silver	14
5	Germany	silver	10
6	Canada	silver	8
7	Norway	bronze	11
8	Germany	bronze	7
9	Canada	bronze	10

Ceci n'est pas une pipe

Now that we've covered tidy data, there is one more concept that is very common in the `tidyverse` that we should discuss. This is the pipe (`%>%`) from the `magrittr` package. This is similar to the Unix pipe, and it takes the left-hand side of the pipe and applies the right-hand side function to it. Take the following code:

```
| mpg %>% summary()
```

The preceding code is equivalent to the following code:

```
| summary(mpg)
```

As another example, look at the following code:

```
| gapminder %>% filter(year > 1960)
```

The preceding code is equivalent to the following code:

```
| filter(gapminder, year > 1960)
```

Piping greatly enhances the readability of code that requires several steps to execute. Take the following code:

```
| x %>% f %>% g %>% h
```

The preceding code is equivalent to the following code:

```
| h(g(f(x)))
```

To demonstrate with a real example, take the following code:

```
| groupedData = gapminder %>%  
|   filter(year > 1960) %>%  
|   group_by(continent, year) %>%  
|   summarise(meanLife = mean(lifeExp))
```

The preceding code is equivalent to the following code:

```
| summarise(  
|   group_by(  
|     filter(gapminder, year > 1960),
```

```
| continent, year),  
| meanLife = mean(lifeExp))
```

Hopefully, it should be obvious which is the easier to read of the two.

Gapminder

Now we've looked at tidying data, let's have a quick look at using `dplyr` and `ggplot` to filter, process, and plot some data. In this section, and throughout this book, we're going to be using the Gapminder data that was made famous by Hans Rosling and the Gapminder foundation. An excerpt of this data is available from the `gapminder` package, as assembled by Jenny Bryan, and it can be installed and loaded very simply using `install.packages("gapminder"); library(gapminder)`. As the package description indicates, it includes, for each of the 142 countries that are included, the values for life expectancy, GDP per capita, and population, every five years, from 1952 to 2007.

In order to prepare the data for plotting, we will make use of `dplyr`, as shown in the following code:

```
groupedData = gapminder %>%
  filter(year > 1960) %>%
  group_by(continent, year) %>%
  summarise(meanLife = mean(lifeExp))
```

This single block of code, all executed in one line, produces a dataframe suitable for plotting, and uses chaining to enhance the simplicity of the code. Three separate data operations, `filter()`, `group_by()`, and `summarise()`, are all used, with the results from each being sent to the next instruction using the `%>%` operator. The three instructions carry out the following tasks:

- `filter()`: This is similar to `subset()`. This operation only keeps rows that meet certain requirements—in this case, years beyond 1960.
- `group_by()`: This allows operations to be carried out on subsets of data points—in this case, each continent for each of the years within the dataset.
- `summarise()`: This carries out summary functions, such as `sum` and `mean`, on several data points—in this case the mean life expectancy within each continent and available year.

So, to summarize, the preceding code filters the data to select only years beyond 1960, groups it by the continent and year, and finds the mean life expectancy within that continent or year. Printing the output from the preceding code yields the following:

```
> groupedData
# A tibble: 50 x 3
# Groups:   continent [?]
  continent year meanLife
  <fct>     <int>    <dbl>
1 Africa      1962    43.3
2 Africa      1967    45.3
3 Africa      1972    47.5
4 Africa      1977    49.6
5 Africa      1982    51.6
6 Africa      1987    53.3
7 Africa      1992    53.6
8 Africa      1997    53.6
9 Africa      2002    53.3
10 Africa     2007    54.8
# ... with 40 more rows
```

As you can see, the output is a tibble, which has a nice `print` method that only prints the first several rows. Tibbles are very similar to dataframes, and are often produced by default instead of dataframes within the `tidyverse`. There are some nice differences, but they are fairly interchangeable with dataframes for our purposes, so we will not get sidetracked by the differences here.

Now we have mentioned tibbles, you can see that the dataframe is a nice summary of the mean life expectancy by year and continent.

A simple Shiny-enabled line plot

We have already seen how easy it is to draw line plots in `ggplot2`. Let's add some Shiny magic to a line plot now. This can be achieved very easily indeed in RStudio by just navigating to File | New File | R Markdown | New Shiny document and installing the dependencies when prompted. Once a title has been added, this will create a new R Markdown document with interactive Shiny elements. R Markdown is an extension of Markdown (see daringfireball.net/projects/markdown/), which is itself a markup language, such as HTML or LaTeX, which is designed to be easy to use and read. R Markdown allows R code chunks to be run within a Markdown document, which renders the contents dynamic. There is more information about Markdown and R Markdown in [Chapter 2, Shiny First Steps](#). This section gives a very rapid introduction to the type of results possible using Shiny-enabled R Markdown documents.

For more details on how to run interactive documents outside RStudio, refer to [go.o.g1/Ngubdo](#). By default, a new document will have placeholder code in it which you can run to demonstrate the functionality. We will add the following:

```
---
title: "Gapminder"
author: "Chris Beeley"
output: html_document
runtime: shiny
---

```{r, echo = FALSE, message = FALSE}

library(tidyverse)
library(gapminder)

inputPanel(
 checkboxInput("linear", label = "Add trend line?", value = FALSE)
)

draw the plot
renderPlot{

 thePlot = gapminder %>%
 filter(year > 1960) %>%
 group_by(continent, year) %>%
 summarise(meanLife = mean(lifeExp)) %>%
 ggplot(aes(x = year, y = meanLife,
 group = continent, colour = continent)) +
 geom_line()

 if(input$linear){
```

```
 thePlot = thePlot + geom_smooth(method = "lm")
 }
 print(thePlot)
}
```

```

The first part between the `---` is the YAML, which performs the setup of the document. In the case of producing this document within RStudio, this will already be populated for you.

R chunks are marked as shown, with ````{r}` to begin and ````` to close. The echo and message arguments are optional—we use them here to suppress output of the actual code and any messages from R in the final document.

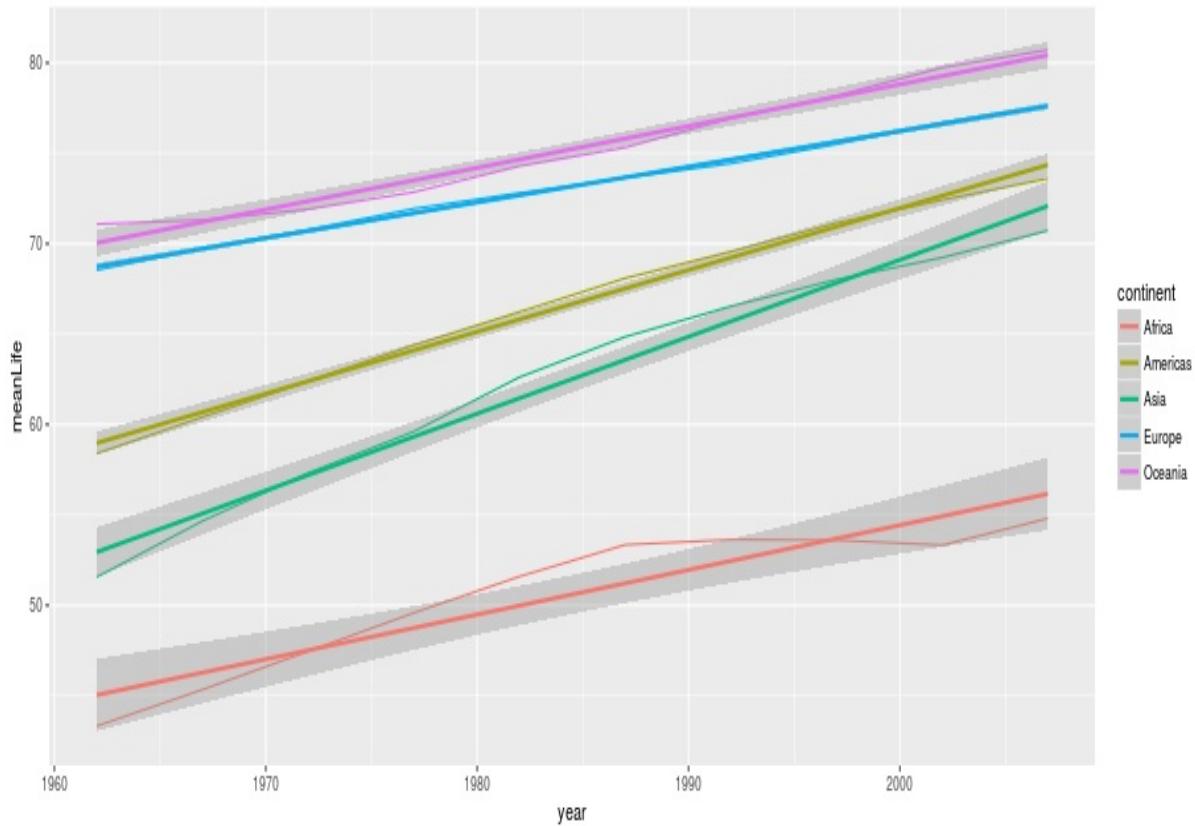
We'll go into more detail about how Shiny inputs and outputs are set up later on in the book. For now, just know that the input is set up with a call to `checkboxInput()`, which, as the name suggests, creates a checkbox. It's given a name ("linear") and a label to display to the user ("Add trend line?"). The output, being a plot, is wrapped in `renderPlot()`. You can see the checkbox value being accessed on the `if(input$linear){...}` line. Shiny inputs are always accessed with `input$` and then their name, so, in this case, `input$linear`. When the box is clicked—that is, when it equals `TRUE`—we can see a trend line being added with `geom_smooth()`. We'll go into more detail about how all of this code works later in the book; this is a first look so that you can start to see how different tasks are carried out using R and Shiny.

You'll have an interactive graphic once you run the document (click on Run document in RStudio or use the `run()` command from the `rmarkdown` package), as shown in the following screenshot:

Gapminder

Chris Beeley

Add trend line?



As you can see, Shiny allows us to turn on or off a trend line courtesy of `geom_smooth()` from the `ggplot2` package.

Installing Shiny and running the examples

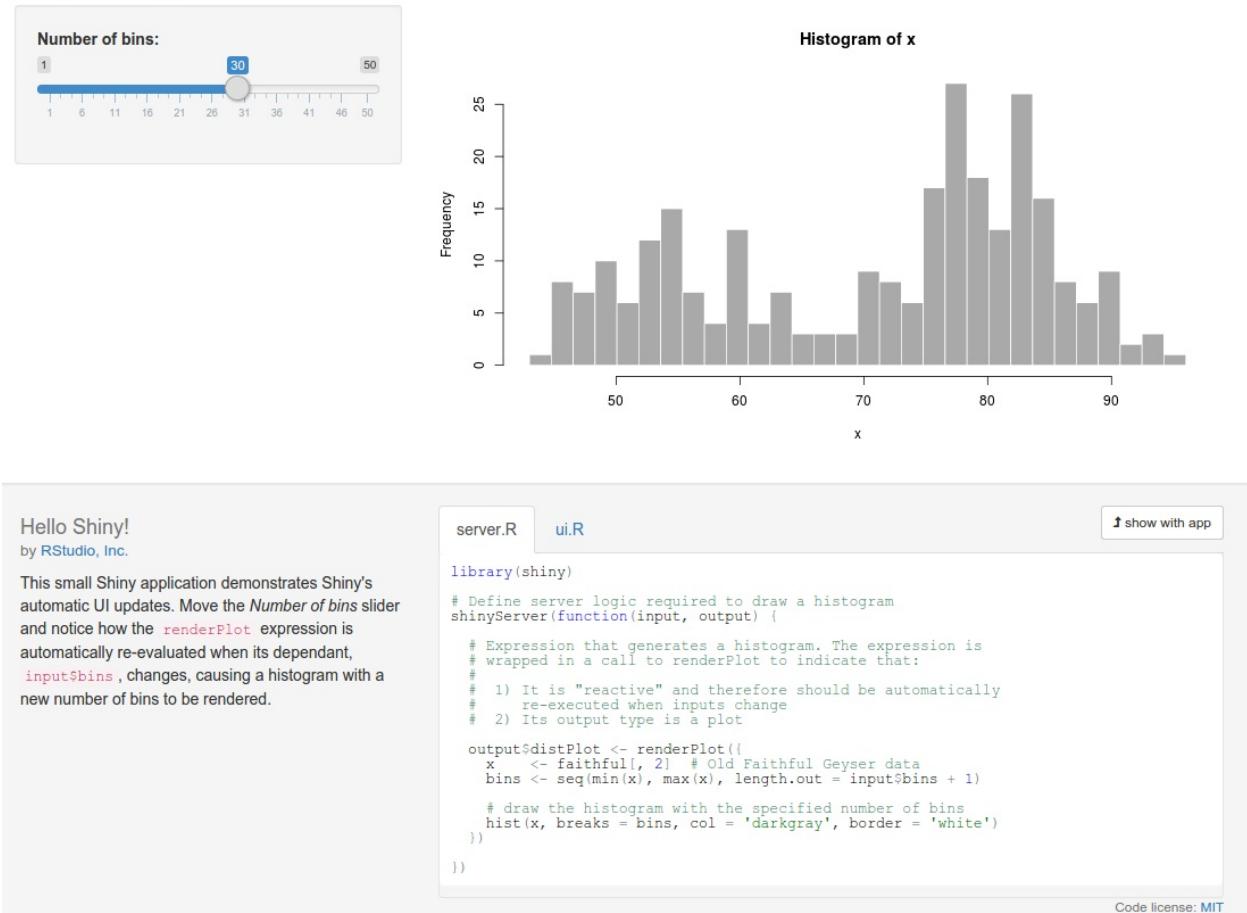
Shiny can be installed using standard package management functions, as described previously (using the GUI or running `install.packages("shiny")` at the console).

Let's run some of the examples, as shown in the following code:

```
|> library(shiny)  
|> runExample("01_hello")
```

Your web browser should launch and display the following screenshot (note that I clicked on the show below button on the app to better fit the graphic on the page):

Hello Shiny!



The graph shows the frequency of a set of random numbers drawn from a statistical distribution known as the normal distribution, and the slider allows users to select the size of the draw, from 0 to 1,000. You will note that when you move the slider, the graph updates automatically. This is a fundamental feature of Shiny, which makes use of a reactive programming paradigm.

This is a type of programming that uses reactive expressions, which keep track of the values on which they are based. These values can change (they are known as reactive values) and update themselves whenever any of their reactive values change. So, in this example, the function that generates the random data and draws the graph is a reactive expression, and the number of random draws that it makes is a reactive value on which the expression depends. So, whenever the number of draws changes, the function re-executes.



You can find more information about this example, as well as a comprehensive tutorial for Shiny, at shiny.rstudio.com/tutorial/.

Also, note the layout and style of the web page. Shiny is based by default on the Bootstrap theme (see getbootstrap.com/). However, you are not limited by the styling at all, and can build the whole UI using a mix of HTML, CSS, and Shiny code.

Let's look at an interface that is made with bare-bones HTML and Shiny. Note that in this and all subsequent examples, we're going to assume that you run `library(shiny)` at the beginning of each session. You don't have to run it before each example, except at the beginning of each R session. So, if you have closed R and have come back, then run it on the console. If you can't remember, run it again to be sure, as follows:

```
| library(shiny)  
| runExample("08_html")
```

And here it is, in all its customizable glory:

HTML UI

Distribution type:

Normal

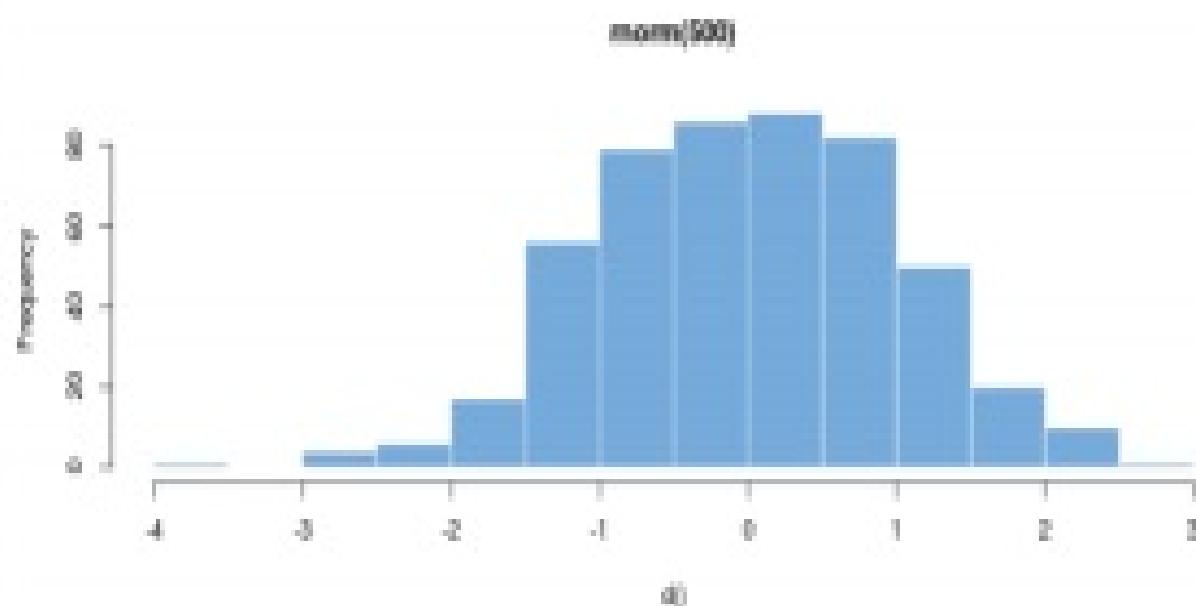
Number of observations:

500

Summary of data:

| Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|-----------|-----------|-----------|----------|----------|----------|
| -0.710217 | -0.713367 | -0.007828 | 0.004377 | 0.730581 | 2.560307 |

Plot of data:



Head of data:

x

-1.31

-1.04

Now, there are a few different statistical distributions to pick from and a different method of selecting the number of observations. By now, you should be looking at the web page and imagining all the possibilities there are to produce your own interactive data summaries and style them just how you want, quickly and simply. By the end of the next chapter, you'll have made your own application with the default UI, and by the end of the book, you'll have complete control over the styling and be pondering where else you can go.

There are lots of other examples included with the Shiny library; just type `runExample()` in the console to be provided with a list.

To see some really powerful and well-featured Shiny applications, take a look at the showcase at shiny.rstudio.com/gallery/.

Summary

In this chapter, we installed R and explored the different options for GUIs and IDEs, and looked at some examples of the power of R. We saw how R makes it easy to manage and reformat data and produce beautiful plots with a few lines of code. You also learned a little about the coding conventions and data structures of R. We saw how to format a dataset and produce an interactive plot in a document quickly and easily. Finally, we installed Shiny, ran the examples included in the package, and were introduced to a couple of basic concepts in Shiny.

In the next chapter, we will go on to build our own Shiny application using the default UI.

Shiny First Steps

In the previous chapter, we looked at R, learned some of its basic syntax, and saw 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 explore the Gapminder data described in the previous chapter. We will cover the following topics:

- The types of Shiny application—R Markdown, single-file, two-file, Shiny gadgets
- Interactive Shiny documents in R Markdown
- Single-file Shiny applications
- Two-file Shiny applications
- A minimal example of a full Shiny application
- Widget types
- The basic structure of a Shiny program
- The selection of simple input widgets (checkboxes and combo buttons)
- The selection of simple output types (rendering plots and maps, and returning text)
- The selection of simple layout types (page with sidebar and tabbed output panel)
- Reactive objects
- A brief summary of more advanced layout features

Types of Shiny application

In the first edition of this book, which was based on Shiny 0.6, we described only two types of application. First, a fairly simple Bootstrap-themed interface with input widgets down the left and output (a single page or a tabbed output window) on the right. The second type consisted of custom-built web pages with their own HTML and CSS files. Shiny has developed quite a bit since then, and there are actually many types of Shiny application and many ways of building them. These are as follows:

- Interactive markdown documents with Shiny widgets embedded
- Shiny applications (default CSS, written entirely in R)
- Web pages (for example, custom CSS, HTML, and JavaScript)
- Shiny gadgets
- Flex dashboards

In this chapter, we will be considering the first two: interactive documents and full applications. [Chapter 3, Integrating Shiny with HTML](#), will cover how to build your own web pages containing Shiny applications. Shiny gadgets are tools for R programmers rather than for end users, and they allow R users to explore data and generate graphics and summaries with Shiny interfaces. They will be described further in [Chapter 7, Power Shiny](#). Flex dashboards will be looked at in [Chapter 6, Dashboards](#).

Interactive Shiny documents in RMarkdown

As we saw in the previous chapter, interactive documents can be made very easily using R Markdown in RStudio. Even if you are not using RStudio, it is a simple matter of writing an R Markdown file with Shiny code in it. If you do not use RStudio, you will need an up-to-date version of Pandoc (the version in many Linux distributions is not recent enough). For more on installing Pandoc on Linux, Windows, or Mac, go to pandoc.org/installing.html.

RMarkdown is based on Markdown, which is a markup language designed to be easily converted into HTML, but which looks much more like a natural document in its raw format, as opposed to HTML or other markup languages (such as LaTeX), which have more prominent and strange-looking tags. For example, the Markdown syntax for a bulleted list is as follows:

```
* First bullet
* Second bullet
* Third bullet
```

The HTML equivalent is as follows:

```
<ul>
  <li>First bullet</li>
  <li>Second bullet</li>
  <li>Third bullet</li>
</ul>
```

Tagging markup in LaTeX is even more verbose. R Markdown uses Markdown conventions, but allows code chunks of R to be run within the document, and allows text and graphical output to be generated from those chunks. Coupled with Pandoc (the Swiss Army knife of document rendering), Markdown and R Markdown can be rendered into many formats, including XHTML, HTML, epub, LaTeX, .pdf, .doc, .docx, and .odt.

R Markdown with Shiny goes one step further and allows users to interact with the document on a web page.

Let's build a minimal example. If you are using RStudio, you will be given a boilerplate Shiny Markdown document to work from, which makes things a bit easier, but here we'll ignore that and build it from scratch. The code is available at goo.gl/N7Qkv8.

Let's go through each part of the document. Navigate to File | New File | R Markdown | New document and enter the following code:

```
# Example RMarkdown document
This is an interactive document written in *markdown*. As you can see it is easy to incl
1. Ordered lists
2. *Italics*
3. **Bold type**
4. Links to [Documentation](http://example.com/)
## This is heading two
Perhaps this introduces the visualisation below.
```

This is the document part of the Shiny document, written in Markdown. The following conventions can be noted:

- The # character is used for headings at level 1, and ## for headings at level 2
- Numbered (ordered) lists are designated with 1, 2, and so on
- Italics are given with *single asterisks* and a bold format is given with **double asterisks**
- Links are represented using the format of (<http://example.com/>)

Next follows a code chunk, beginning with `r` and ending with ``. The echo=FALSE argument is added to the chunk to prevent the printing of the R code. You will usually want to do this, but not on every occasion—for example, when producing a teaching resource:

```
| ``{r, echo=FALSE} sliderInput("sampleSize", label = "Size of sample", min = 10, max = 1
```

Straight away, we can see some of the design principles in Shiny applications. We can see the separation of input code, `sliderInput()`, and output code, `renderPlot()`. The `sliderInput()` function, as the name suggests, defines an input widget that allows the user to select from a range of numeric values, in this case, between 10 and 100, with a starting value of 50 and a step increase of 1. The `renderPlot()` function produces a reactive plot using whatever functions it finds within itself (in this case, the graphical function `hist()`, which draws a histogram).

As we already covered in [Chapter 1](#), *Beginning R and Shiny*, reactive outputs change when their inputs change. The `runif(n)` function produces n random numbers between 0 and 1 (with default arguments). As we can see in this case, n is given by `input$sampleSize`. Inputs are accessed very simply in Shiny in this format; you can see that we named the input `sampleSize` within the `sliderInput()` function, which places the selected value from the widget in `input$sampleSize` (naming it `myInput` places the value in `input$myInput`).

Therefore, `runif()` generates random numbers in the quantity of `input$sampleSize`, `hist()` plots them with a histogram, and `renderPlot({})` tells Shiny that the output within is reactive and should be updated whenever its inputs (in this case, just `input$sampleSize`) change.

The final result will look like the following screenshot:

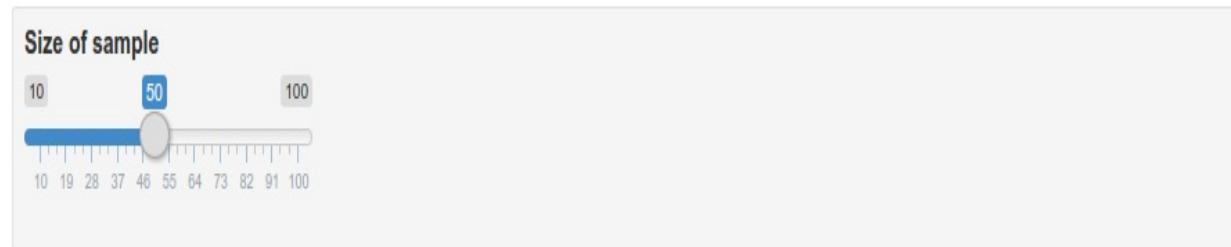
Example RMarkdown document

This is an interactive document written in *markdown*. As you can see it is easy to include:

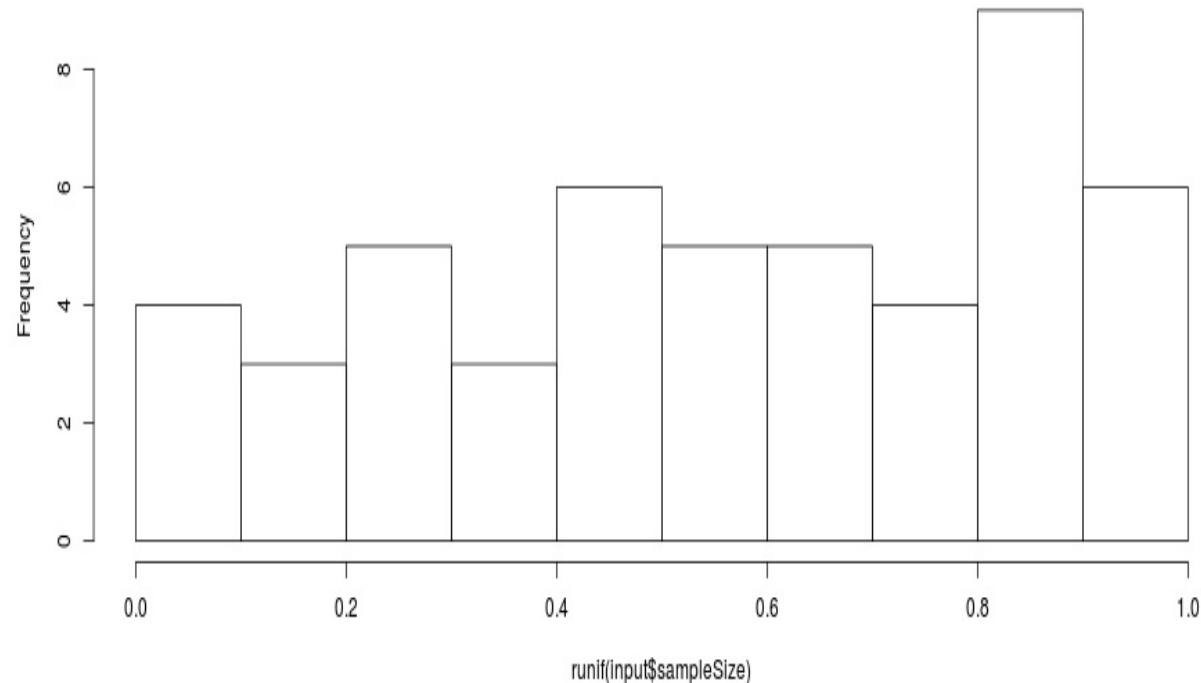
1. Ordered lists
2. *Italics*
3. **Bold type**
4. Links to Documentation

This is heading two

Perhaps this introduces the visualisation below.



Histogram of `runif(input$sampleSize)`



That's it! You made your first Shiny application. It's that easy. Now, let's consider building fully fledged applications, starting with a minimal example and building up from there.

A minimal example of a full Shiny application

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`.

The ui.R of the minimal example

The `ui.R` file is a description of the UI, and is often the shortest and simplest part of a Shiny application. In the following code, 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 the humans producing the code:

```
fluidPage(                                     # Line 1
  titlePanel("Minimal application"),          # Line 2
  sidebarLayout(                             # Line 3
    sidebarPanel(                            # Line 4
     textInput(inputId = "comment",           # Line 5
                 label = "Say something?",     # Line 6
                 value = "")                  # Line 7
    )),
  mainPanel(                                # Line 9
    h3("This is you saying it"),            # Line 10
    textOutput("textDisplay")               # Line 11
  )
)
```

The following list is an explanation of each line:

- Line 1: Flexible layout function
- Line 2: Title
- Line 3: Standard inputs on the sidebar; outputs in the main area layout
- Line 4: The sidebar layout function
- Line 5: Give the name of the input element; this will be passed to `server.R`
- Line 6: The display label for the variable
- Line 7: The initial value
- Line 9: The output panel
- Line 10: The title drawn with the HTML helper function
- Line 11: The output text with the ID, `textDisplay`, 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 folder R's working directory (using the `setwd()` command—for example, `setwd("~/shinyFiles/minimalExample")`, or with the `session > Set working directory` menu option)

3. Load the Shiny package with the `library(shiny)` command
4. Type `runApp()` in the console (or, in Rstudio, click Run app just above the code window)

Using `runApp()` with the name of a directory within works just as well—for example, `runApp("~/shinyFiles/minimalExample")`. So instead of setting the working directory to the location of your application and then using `runApp()` separately, the whole thing can simply be carried out in one instruction, passing `runApp()` in the name of the directory directly. Just remember that it is a directory and not a file that you need to point to.

The first instruction, `fluidPage(...)`, tells Shiny that we are using a fluid page layout. This is a very flexible layout function whose functionality we will explore further in [Chapter 4, Mastering Shiny's UI Functions](#). Next, the title of the application is defined very simply using the `titlePanel()` function. Then follows the main layout instruction; in this case, we are going to use the simplest UI layout, `sidebarLayout()`, which places inputs on the left (or right, optionally) and the main output section in the middle. All of the UI elements are defined within the `sidebarLayout()` function.

The next two instructions perform the main UI setup, with `sidebarPanel()` setting up the application controls and `mainPanel()` setting up the output area.

The `sidebarPanel()` phrase will usually contain all of the input widgets; in this case, there is only one: `textInput()`. The `textInput()` widget 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 start 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 that I used one of the HTML helper functions to make a little title, `h3("...")`. There are many of these helper functions included, and they are incredibly useful for situations where you either don't want to do too much styling with HTML and CSS yourself or don't know how. Let's just stop very quickly to look at a few examples.

A note on HTML helper functions

There are several HTML helper functions that are designed to generate HTML to go straight on the page; type `?p` in the console for the complete list. These functions allow you to mark up text in HTML using R code—for example, `h3("Heading 3")` will produce `<h3>Heading 3</h3>`, `p("Paragraph")` will produce `<p>Paragraph</p>`, and so on.

The HTML tags that will be available using this function include `
`, `<code>`, `<div>`, ``, `<h1>`, ``, `<p>`, `<pre>`, and ``. Even more tags are available through the use of the `tags()` function. There is more on Shiny and HTML in [Chapter 3, *Integrating Shiny with HTML*](#), and a full list of tags and other help is available in the documentation at shiny.rstudio.com/articles/html-tags.html.

The finished interface

The other element that goes in `mainPanel()` is an area to handle reactive text that is 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:

Minimal example

A screenshot of a Shiny application interface. On the left, there is a text input field with the placeholder "What would you like to say?". Below it is a text area containing the text "Hello, world!". On the right, the text "This is you saying it" is displayed in bold. Below that, a smaller text states "You said 'Hello, world!'. There are 13 characters in this."

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`, which knows where they all go and what types of output they are.

The server.R of the minimal example

Let's now look at `server.R`, where it should all become clear. Look at the following code:

```
function(input, output) { # server is define      d here output$textDisplay = renderText({  
  paste0("You said '", input$comment, # from text input  
  "' . There are ", nchar(input$comment),  
  " characters in this.")  
})  
}
```

We define the reactive components of the application within `function(input, output) {...}`. 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$textDisplay = renderText({...})`. This instruction does two basic things. First, it gives the output a name (`textDisplay`) so that it can be referenced in `ui.R` (you can see it in the last part of `ui.R`). Second, it tells Shiny that the content contained within is reactive (that is, it will be updated when its inputs change) and it takes the form of text. We will 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, available at 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. The full code can be found here, [h](#)

<https://gist.github.com/ChrisBeeley/4202605cf2e64b4f609e>. Using these very simple building blocks, you can actually make some really useful and engaging applications.

The program structure

Since the first edition of this book, a significant change has taken place with regards to how Shiny applications are structured. A new feature has been added, giving us the ability to place them all within one code file. This is most useful when building small demonstrations or examples for other users, who can just paste the whole code file into the console and have the application run automatically. In order to make use of this functionality, just combine the code from `server.R` and `ui.R`, as shown in the following example:

```
library(shiny)
server <- function(input, output) {
  #contents of server.R file
}
ui <- fluidPage( # or other layout function
  # contents of ui.R file
)
shinyApp(ui = ui, server = server)
```

This is useful neither for large applications, nor for the purposes of explaining the functions of particular parts of code within this book, so we shall ignore it from now on. Just be aware that it's possible; you may well come across it on forums, and you may wish to contribute some small examples yourself.

An 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 and 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$comment`. The `server.R` file defines a reactive expression, `renderText()`. It depends on `input$comment`.

Note that this dependence is defined automatically by Shiny.

The `renderText()` expression uses an output from `input$comment`, so Shiny automatically connects them. Whenever `input$comment` changes, `renderText()` will automatically run with the new value. The optional exercise gave two reactive values to the `renderPlot()` call, and so, whenever either changes, `renderPlot()` will be 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 capabilities that Shiny offers and when to use them.

Embedding applications in documents

To briefly return to the subject of interactive documents, it is worth noting that it is possible to embed entire Shiny applications within interactive documents rather than having the rather stripped-down functionality that we embedded within a document earlier in the chapter. Just include a link to the directory that holds the application, like this:

```
```{r, echo=FALSE}
shinyAppDir(
 "~/myApps/thisApplication",
 ...)
```

For more information about embedding, type `?shinyApp` in the console.

# 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 their outputs and the type of data they return. To run it, just enter the following command:

```
| > library(shiny)
| runGist(6571951)
```

This is one of the several built-in functions of Shiny that allow you to run code hosted on the internet. Details about sharing your own creations in other ways are discussed in [Chapter 9, Persistence, Storage, and Sharing](#). The finished application looks like the following screenshot:

# Widget values and data types

1. checkboxGroupInput

Ice cream  
 Trifle  
 Pistachios

2. checkboxInput

3. dateInput

2013-09-15

4. dateRangeInput

2013-08-25 to 2013-09-27

5. numericInput

6

6. radioButtons

Taxi  
 Take a walk

7. selectInput

Situation comedy ▾

8. sliderInput

1 7 10

9. textInput

Hello, world!

## Output and data type

	Value	Class
1	IC,Trifle	character
2	TRUE	logical
3	2013-09-15	Date
4	2013-08-25 2013-09-27	Date
5	6	numeric
6	Walk	character
7	Sitcom	character
8	7	numeric
9	Hello, world!	character

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

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

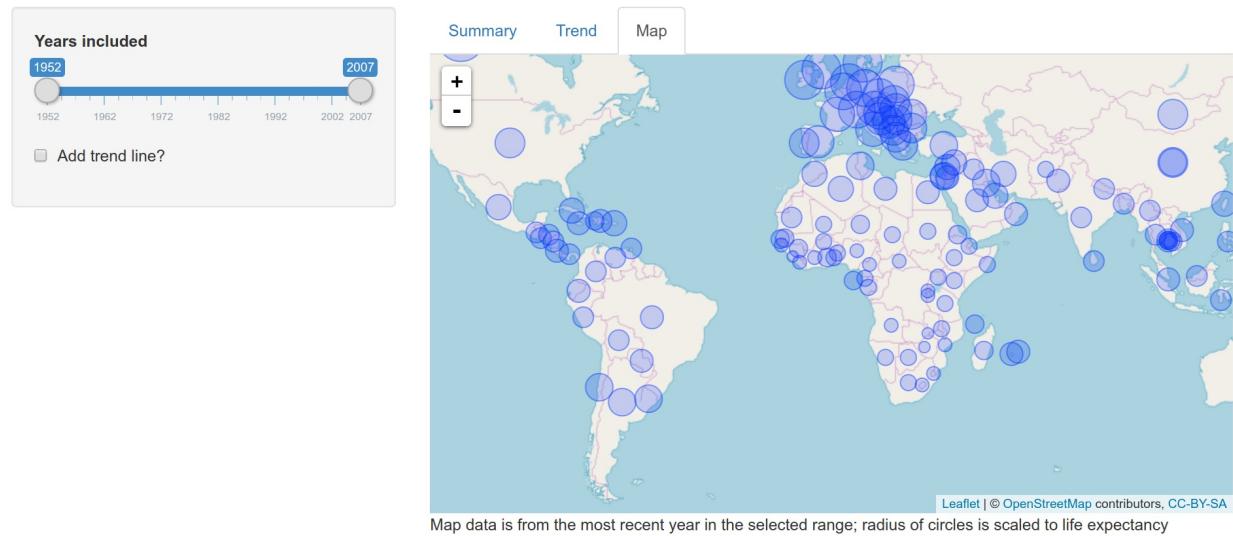
# The Gapminder application

Now that we've got the basics, let's build a full application. Before we proceed, note that we will need to install a few packages—`tidyverse`, `gapminder`, `leaflet`, and `ggmap`. Each can be installed from CRAN (the official R package repository) using the code phrases `install.packages("tidyverse")`, `install.packages("gapminder")`, and so on. We will not install `ggmap` this way, though. At the time of writing, there is a bug in the CRAN version. We'll install the `dev` version instead, as shown in the following code:

```
| install.packages("devtools")
| library(devtools)
| devtools::install_github("dkahle/ggmap")
```

The application is pretty simple to get us started, but it illustrates several important methods and principles in Shiny. It features tabbed output, which allows the user to select different inputs or groups, which are each kept on a separate tab. It features the standard Shiny layout—the sidebar layout—with inputs at the left and outputs in the main section. The three tabs give a textual summary, a line graph showing life expectancy over time, and a map with circles scaled to the life expectancy in each country. The application looks like the following screenshot:

# Gapminder



# The UI

If you can, download and run the code and data from [github.com/ChrisBeeley/gapminder](https://github.com/ChrisBeeley/gapminder) (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 that you can play with it), just use another function to share and run applications (we will discuss this in [Chapter 5, Easy JavaScript and Custom JavaScript Functions](#)), as shown in the following code:

```
| runGitHub("gapminder", "ChrisBeeley")
```

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

```
library(shiny)
library(leaflet)

fluidPage(
 titlePanel("Gapminder"),
```

The first thing that we do is load the `leaflet()` package. The `leaflet()` package is used for drawing interactive maps. We'll discuss it in detail later. For now, just note that it is necessary to load it in the UI in order to access a special output function that doesn't exist within Shiny—`leafletOutput()`. The whole interface is wrapped in `fluidPage()`. This is true of most, but not all, Shiny applications. We will look in detail at the basic types of Shiny page and setup in [Chapter 4, Mastering Shiny's UI Functions](#). It performs the basic setup of the page, making it responsive and ensuring that it looks good on differently sized browsers. There is more detail on this function in [Chapter 4, Mastering Shiny's UI Functions](#). The `titlePanel()` function is used to give the application a nice big title.

The next section shows the setup of the standard layout of a Shiny application. Simplified, it looks like the following code, with `sidebarPanel()` defining the inputs on the left and `mainPanel()` the inputs on the right, all wrapped in `sidebarLayout()`:

```
sidebarLayout(
 sidebarPanel(
 # input controls in here
```

```
),
 mainPanel(
 # outputs here
)
```

Do note that keeping the inputs on the left and the outputs on the right is the typical layout, but Shiny is totally flexible. You can keep some inputs on the right if you wish, or even outputs on the left. This is just the usual setup that you will see in a simple Shiny application.

Within `sidebarPanel()`, we find two inputs, `sliderInput()` and `checkboxInput()`, as shown in the following code:

```
sliderInput(inputId = "year",
 label = "Years included",
 min = 1952,
 max = 2007,
 value = c(1952, 2007),
 sep = "",
 step = 5
),
checkboxInput("linear", label = "Add trend line?", value = FALSE)
```

The `sliderInput()` function allows your user to select a number, or a range, using a slider. In this case, they will be selecting a range, the range of years that they are interested in for the text summary and the time series graph. We have already seen the first two arguments, and they will become very familiar to you—`inputId` giving the input a name so it can be referred to as `input$name` (`input$year` in this case) and `label`, which gives the control a nice friendly label for the user to read so they can understand the application. You can see the other arguments, giving a minimum, a maximum, and an initial value (in this case, two, to define a range). We can optionally define the separator to separate the thousands in the numbers—in this case, setting it to nothing in order to stop the years appearing as 1,952, 2,007, and so on. Note the use of `step = 5`. This gives the numeric gap between each notch on the slider. If it is left as `NULL`, then Shiny will pick something sensible. We use `5` here because the Gapminder data is split into five-year chunks.

The `checkboxInput()` function very simply gives you a tickbox that returns `TRUE` when ticked and `FALSE` when unticked. This example includes all the possible arguments, which gives it a name and label and selects the initial value.

This concludes the inputs. Let's look at the output panel, as shown in the following code:

```
), # end of sidebarPanel()

mainPanel(
 tabsetPanel(
 tabPanel("Summary", textOutput("summary")),
 tabPanel("Trend", plotOutput("trend")),
 tabPanel("Map", leafletOutput("map"),
 p("Map data is from the most recent year in the selected range;
 radius of circles is scaled to life expectancy"))
)
)
```

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`. As you can see, there are three types of output given by three different output functions—text, shown by `textOutput()`; a plot, shown by `plotOutput()`; and a map produced with the `leaflet` package and shown with `leafletOutput()` (more on which later).

# 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, and is even where some of the functions that handle advanced UI features live. Let's look at the chunks in order and talk about the work carried out in each section.

The first chunk of code looks like the following:

```
library(tidyverse)
library(gapminder)
library(leaflet)
library(ggmap)
```

You can see the packages that we need being loaded at the top. We load the `tidyverse` for access to things such as `dplyr` and `ggplot`, for munging data and plotting, respectively. The `gapminder` package is used to load a subset of the `gapminder` data. It has been very kindly munged and placed online in the form of a CRAN package by Jenny Bryan. We will talk more about the data that it makes available later. The `leaflet` package—which, as we mentioned before, is used for drawing maps—is loaded next. Finally, the `ggmap` package is loaded. This will be used to convert country names to latitude and longitude for plotting by `leaflet`.

The next thing that we do is munge or, if we already munged it, load the data, as shown in the following code:

```
if(!file.exists("geocodedData.Rdata")){
 mapData = gapminder %>%
 mutate(country2 = as.character(country)) %>%
 group_by(country) %>%
 slice(1) %>%
 mutate_geocode(country2, source = "dsk") %>%
 select(-country2)

 mapData = left_join(gapminder, mapData) %>%
 group_by(country) %>%
 fill(lon) %>%
 fill(lat)

 save(mapData, file = "geocodedData.Rdata")
} else {
```

```
| } load("geocodedData.Rdata")
```

I'm not going to distract from Shiny by talking too much about this code. Essentially, it checks to see whether the data is already in the relevant directory (which it will be after you've run it once). If it is, it just loads the data. If not, it prepares it by producing a smaller dataset with one instance of each country and geocoding it (that is, turning it into latitude and longitude), then combining it with the whole dataset (to put all the years for each country back in), and filling in the resulting missing geocodings using the incredibly useful `fill()` function of `dplyr`. Having done all that, it saves the data for next time (since querying the API for all 142 countries takes quite a long time). Don't worry too much if you can't follow this code at the moment; it's really there just to get the data onto your computer in a simple way.

It's worth noting that data instructions—and indeed any other code—that appear above `function(input, output) {}` are executed once when the application starts and then serve all instances of the Shiny application. This makes no difference when you're developing on a local machine, since the code will start fresh every time you run it, but once you move your code to a server, it can make a big difference, since the code is only run once for all the users and application instances you have (until you restart the Shiny server, which manages connections to your applications on a server, of course). Therefore, any intensive data or processing calls are best kept in this section to avoid your users having to wait a long time for their application to load.

# Reactive objects

The rest of this code is wrapped in `function(input, output){}`. This is the reactive part of your application. We will talk in more detail about reactive programming later in the book; for now, let's just say that reactive programming is a type of programming where when the inputs change, the outputs change.

The next piece of code looks like this:

```
theData = reactive({
 mapData %>%
 filter(year >= input$year)
})
```

This code defines a reactive object. Up until now, the `server.R` file has just contained a list of output commands that produce the output, ready to fill the allocated spaces in `ui.R`. Here, we're working a little differently. 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 that use the same dataset (as in this case), 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 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. Crucially, when it runs, its output is cached. This means that as long as its inputs don't change, it will not rerun if it is called on again by a different part of your application. This prevents your application from running the same data processing tasks repeatedly.

In this case, the data processing is very small, so we're not really saving any time

using reactive objects; however, it is still good practice to use them because, as we just mentioned, it means that you only have one data function to maintain rather than several scattered between the outputs.

Let's have a look at an example:

```
| theData = reactive({
| mapData %>%
| filter(year >= input$year[1], year <= input$year[2])
| })
```

This function, very simply, filters the data so that it contains only data that was logged between the years selected in `sliderInput()`. 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 *reactive* by enclosing it in `reactive({...})` and assigning it, very simply, to `theData`. This generates a reactive object named `theData`. This can be accessed by calling this function—that is, by running `theData()` (for the whole dataframe), or `theData()$variableName` (for a variable), or `theData()[, 2:10]` (for the second to the tenth variable). Note the brackets after `theData`. More information on reactive objects, when to use them, and lots of advice about managing and controlling reactivity and data processing in your application is given in [Chapter 8, \*Code Patterns in Shiny Applications\*](#).

# Outputs

Finally, we will look at how the outputs are defined. Let's look first at the code that produces the first tab of output, which is the textual summary of the data.

# Text summary

The text summary function is wrapped in `renderText()`. This lets Shiny know that this function returns text, as shown in the following code:

```
output$summary = renderText({
 paste0(input$year[2] - input$year[1], " years are selected. There are ",
 length(unique(theData()$country)), " countries in the dataset measured at ",
 length(unique(theData()$year)), " occasions.")
})
```

Remember that this output is picked up in `ui.R` with the `textOutput()` function, which takes the "summary" identifier, which, as you can see, is the name given by the `output$summary = renderText({...})` function. The function very simply pastes together some text and some values taken from the application.

# Trend graphs

The next part of the code specifies the graph of the trend in life expectancy using `ggplot2`. Let's look at each piece of code:

```
trend
output$trend = renderPlot({
 thePlot = theData() %>%
 group_by(continent, year) %>%
 summarise(meanLife = mean(lifeExp)) %>%
 ggplot(aes(x = year, y = meanLife, group = continent, colour = continent)) +
 geom_line() + ggtitle("Graph to show life expectancy by continent over time")
 if(input$linear){
 thePlot = thePlot + geom_smooth(method = "lm")
 }
 print(thePlot)
})
```

The first line defines the output as a reactive plot. The second instruction uses chained `dplyr` instructions, as we saw in [Chapter 1, Beginning R and Shiny](#), first to group the data by continent and year, and then to calculate the mean life expectancy in the groups that result (in Africa in each year, in America in each year, and so on). This is then sent on to a `ggplot` instruction for the given year on the `x` axis, mean life expectancy on the `y` axis, and the groupings/colors defined by the continent. Note that by assigning it to `thePlot`, we do not print it, but merely begin to build it up.

The next section tests for the value of the smoothing checkbox, `input$linear`, and if it is `TRUE`, a regression line is added to the plot. The requirement to `print()` `ggplot` graphics has been dropped from Shiny (in the first edition of this book, it was necessary for you to `print()` each graphic). This graphic will need to be printed in any environment, even the console, because it has not been called with `ggplot()`, but merely assigned to `thePlot`. Normally, when using `ggplot()` directly in a Shiny session, there will be no need to `print()` the plot.

# A map using leaflet

Finally, we add a map to the last tab using the leaflet package. The leaflet package is an R interface to the excellent JavaScript leaflet package, which can be used to build a wide variety of maps from many different sources and annotate them with data in a number of different ways. For more details on the leaflet package, visit [rstudio.github.io/leaflet/](https://rstudio.github.io/leaflet/). The code is relatively simple and looks like the following:

```
output$map = renderLeaflet({
 mapData %>%
 filter(year == input$year[2]) %>%
 leaflet() %>%
 addTiles() %>%
 setView(lng = 0, lat = 0, zoom = 2) %>%
 addCircles(lng = ~lon, lat = ~lat, weight = 1,
 radius = ~lifeExp * 5000,
 popup = ~paste(country, lifeExp))
})
```

It's not essential that you understand this code at this point, and we will be making use of the leaflet package in the rest of the book, but let's look at it line by line to get an idea of how it works:

- Line 1: This defines `output$map` as containing a leaflet map so that Shiny knows how to handle the output.
- Line 2: This tells the function to use the `mapData` that we already loaded right at the top of the file (not, in this case, the reactive data returned by `mapData()`).
- Line 3: This filters the data so that only the most recent data given in the selection can be selected.
- Line 4: This tells R that we want to use a leaflet.
- Line 5: This draws the background to the map—lots of different maps are available. See `?addTiles` for more help on this function.
- Line 6: This defines which bit of the map we are looking at and how zoomed in we are. This will focus on latitude 0 and longitude 0, with the map zoomed most of the way out.
- Lines 7 to 9: These add the actual data points; here, you can see that we give the latitude and longitude as one-sided equations, reduce the weight (pen width), and give the formula to determine the radius of the circles (`life`

expectancy with a 5,000 scaler so it shows at the right size). Finally, we define the `popup`, which is what appears when you click each circle (in this case, the name of the country and the actual life expectancy).

# Advanced layout features

In this chapter, we have covered the most simple of the layout features in Shiny with the help of the `sidebarLayout()`, `mainPanel()`, and `tabsetPanel()` functions. In later chapters, we will build larger and more complex applications, including dashboards, and make use of more advanced layout features. It is worth pausing here briefly to take a quick look at the other types of layout that are available so that you can think about the best way to implement your own application as we go through the next couple of chapters.

There are essentially two more broad types of layout function that you can use in Shiny. The first uses the layout features of Bootstrap and allows you to precisely define the layout of your application using a grid layout. Essentially, Bootstrap asks you to define the UI as a series of rows. Each row can be further subdivided into columns of varying widths.

Each set of columns on a row has widths that add up to `12`. In this way, you can quite easily specify, for example, the first row as consisting of one column of width `12`, and then the second row as consisting of two columns, one of width `2` and one of width `10`. This creates a header panel across the top of the screen, and then a thin one and a thick one two columns below, which you are likely to put UI and output elements in, respectively. A whole variety of layouts is possible, and nesting and offsetting the columns are both possible, which means that with the right code, you can build any grid-based layout you can think of. We will look at the code to implement custom layouts in this way in more detail in later chapters.

If you don't want to set up your UI in that much detail, Shiny provides lots of other layouts that can be very easily called to lay out your application in a particular way. These functions include `navbarPage()`, `navList()`, `verticalLayout()`, and `splitLayout()`. We will look at all of the layout functions in [Chapter 4, \*Mastering Shiny's UI Functions\*](#), but it's worth noting here that there are lots of different ways to control the layout of a Shiny application.

# 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`. You've learned what each part of the code does, including setting up `ui.R` with the position and type of inputs and outputs, and setting up `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. You've learned about the default layout in Shiny, `sidebarLayout()`, as well as the use of `mainPanel()` and `tabsetPanel()`.

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

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

# Integrating Shiny with HTML

So, we built our own application to explore the Gapminder data. You learned about the basic setup of a Shiny application and saw a lot of the widgets. It will be important to remember 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, as we did in the previous chapter, but 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 Gapminder application, or another one of your own (if you have one), using each of the concepts. If you do this, you will have a beautifully styled and interactive application by the end 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.

In this chapter, we are going to cover the following areas:

- Adding HTML to native Shiny applications
- Customizing Shiny applications, or whole web pages, using HTML
- Styling your Shiny application using CSS
- Incorporating Shiny content within another web page
- HTML templates

# 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 the source code and data, can be found on my website at [chrisbeeley.net/website](http://chrisbeeley.net/website). 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 fact, Shiny is very accommodating, and you will find that it will quite happily accept a mix of Shiny code and HTML code produced by you using Shiny helper functions and the raw HTML, also 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 give 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

We will now make a little toy application that shows the life expectancy over time for three different countries, and that gives a link to the Wikipedia page for each. It's not really designed to look good or be useful, but rather to illustrate several things that are possible when you wish to add small amounts of HTML to a native Shiny application, rather than building from scratch or working with HTML templates (more on which later).

We're going to build an application that allows you to select from a couple of different countries and view the life expectancy over time in that country. A custom HTML button shows the Wikipedia page for the selected country.

# ui.R

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

```
fluidPage(
 tags$head(HTML("<link href='http://fonts.googleapis.com/css?family=Jura'
 rel='stylesheet' type='text/css'>")),
 h2("Custom HTML", style = "font-family: 'Jura';
 color: green; font-size: 64px;"),
 sidebarLayout(
 sidebarPanel(
 radioButtons("country", "Country",
 c("Afghanistan", "Bahrain", "Cambodia"))
),
 mainPanel(
 h3("Time series"),
 HTML("<p>Life expectancy over time</p>"),
 plotOutput("plotDisplay"),
 htmlOutput("outputLink")
)
)
)
```

There's a quick method of styling text inline in this example. First, let's fetch a font from Google Fonts, which you can see by simply using the `tags$` function to generate an HTML `<head>` and then placing the link inside, as shown in the following code:

```
tags$head(HTML("<link
 href='http://fonts.googleapis.com/css?family=Jura'
 rel='stylesheet' type='text/css'>"))
```

Now that we have the font available, it's a simple matter of placing it within the application. The `titlePage()` argument, which is often used at the top of a Shiny application, has been removed and replaced with `h2()` because `h2()` will allow you to insert inline styling straight into the function, whereas `titlePage()` will not. Moreover, although the accepted arguments are different, the actual HTML generated by `titlePage()` and `h2()` is the same. To test this yourself, go to the console and type `titlePage("Test")` and then try using `h2("Test")`.

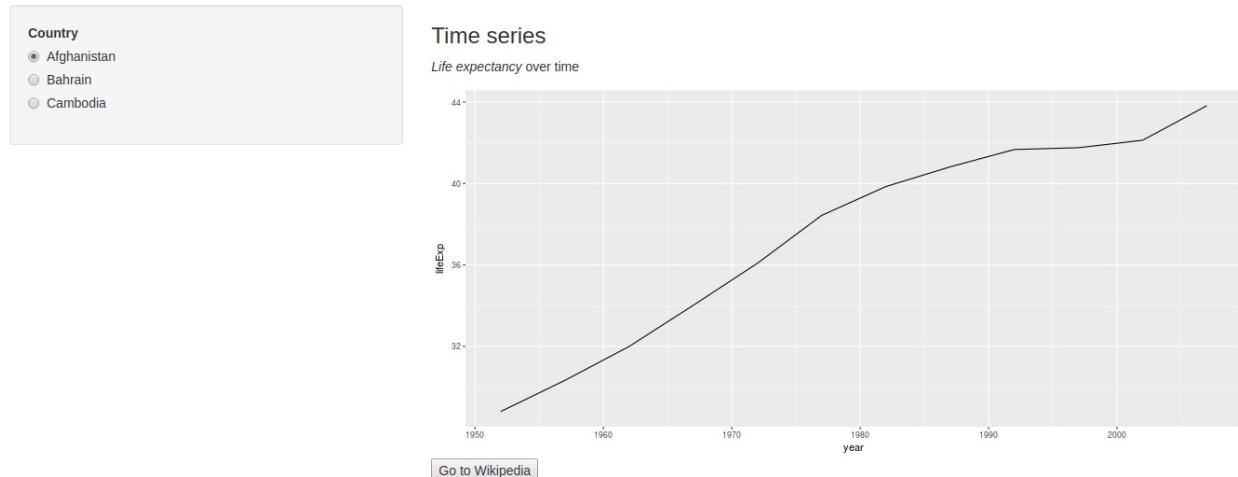
In both cases, the same thing is returned—`<h2>Test</h2>`. This is a really useful

way of learning more about Shiny and helping you to debug more complex applications that make use of Shiny functions, as well as HTML and CSS. Sometimes, it can be necessary to run the application and then inspect the HTML using the inspect source function available in most web browsers (Chrome, Explorer, Firefox, Safari, and so on). Running the function directly in the console is a lot quicker and less confusing. Having done this, it's a simple matter of passing styling information into `h2()`, as you would pass inline styling into a font in HTML:

```
| style = "font-family: 'Jura'; color: green; font-size: 64px;"
```

Also included is the `HTML()` function, which marks text strings as HTML, preventing the HTML from escaping, which would otherwise render this on the screen verbatim. The other new part of this file is the `htmlOutput()` function. This, in a similar way to the `HTML()` function, prevents HTML from escaping and allows you to use your own markup, but this time for text passed from `server.R`. Here's the final interface:

## Custom HTML



# server.R

The `server.R` in this case is quite simple. First, we load the `tidyverse` and the data (the data we created in the previous chapter), as shown in the following code:

```
library(tidyverse)
load("geocodedData.Rdata")
```

Then, we define the reactive part of the application, as shown in the following code:

```
function(input, output) {
 output$plotDisplay <- renderPlot({
 gapminder %>%
 filter(country == input$country) %>%
 ggplot(aes(x = year, y = lifeExp)) +
 geom_line()
 })
 output$outputLink <- renderText({
 link = "https://en.wikipedia.org/wiki/"
 paste0('<form action="", link, input$country, ''>
<input type="submit" value="Go to Wikipedia">
</form>')
 })
}
```

The first part of the code should hold no surprises, being a fairly simple line graph produced using `ggplot2`. You can see the familiar `filter()` function being used to restrict the series to data from one country. The second part of the code uses `renderText()`, which we have used before to show text within an output. The only difference here is that we are going to generate our own HTML. You will recall from the `ui.R` that this output needs to be wrapped in `htmlOutput()`, not `textOutput()`, to preserve the HTML (which would otherwise be automatically escaped in Shiny).

That's a nice gentle introduction to starting to mix HTML into your Shiny applications. Now, let's go to the other extreme and look at building your interface entirely in HTML.

# A 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 named `www` and place a file named `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, as shown in the following code:

```
<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 code for Shiny, loading the JavaScript and jQuery scripts, which make it work, as well as a style sheet 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 jumble at the bottom of the screen, making the whole thing 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, as well as 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>

<textarea name="comment" rows = "4"
 cols = "30">My first graph</textarea>
</p>
<div class="attr-col shiny-input-radiogroup" id="graph">
<p>
 <label>What sort of graph would you like?</label>

 <input type="radio" name="graph" value="1"
 title="Straight line" checked>Linear

 <input type="radio" name="graph" value="2"
 title="Curve">Quadratic

</p>
</div>
<label>Here's a picture of my cats</label>

</div>

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

 <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 inputs 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 to it), 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`). You will note that there is another `<div>` around the radio button definition, `<div class="attr-col shiny-input-radiogroup" id="graph">`. This is a necessary extra when using Shiny with HTML, as discussed at [goo.gl/Lrx9GB](http://goo.gl/Lrx9GB). Another advantage of using your own HTML is that 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 that is named `textDisplay` and picks up `output$textDisplay`, as defined in `server.R`, and the other that is named `plotDisplay` and picks up `output$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 the 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 whatever you like.

# server.R

Let's take a quick look at the `server.R` file, shown in the following code:

```
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 note that the `renderPlot()` function begins by setting the background color to the same as the page itself (`par(bg = "#ecf1ef")`; 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 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 errors when using 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 in 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 because 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:

# Minimal HTML UI

Title for graph:

My first graph

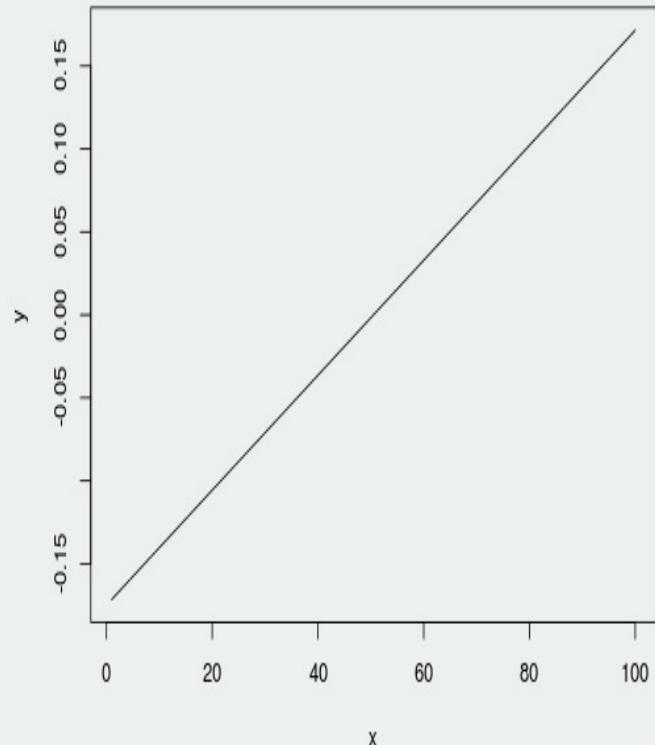
What sort of graph would you like?

- Linear
- Quadratic

Here's a picture of my cats



Title:'My first graph'. There are 14 characters in this.



Finished article

# Including a Shiny app on a web page

The simplest way to mix Shiny and an existing web page is by using an `iframe`. Just like with the RMarkdown document in the previous chapter, it allows you to incorporate an entire Shiny application into an existing document, except in this case, the document is a web page rather than a Markdown document. The only restriction is that you will need to host the application somewhere on the internet so you can point to it. Once you've done that, it's as simple as just pointing to the application, like so:

```
|<iframe src="https://chrisbeeley.net/shinyapps/shinybook3rdedition/chapter2/widgettypes/
```

# HTML templates

The developers of Shiny themselves noted that although it has always been possible to produce entire Shiny applications in HTML, it is very rare to find any examples where somebody has done so. This is probably because of Shiny's ability to produce attractive applications using pure R code (and its ability to incorporate snippets of HTML), as well as the relative complexity of writing the whole interface yourself. To make things simpler, Shiny 0.13 added the ability to use HTML templates. Using an HTML template, you can very easily mix together HTML and Shiny code. There are two main ways of doing this: firstly by defining the code inline within the HTML, and secondly by defining it with the `ui.R` file. In both cases, you will need three files—a `server.R` file, a `ui.R` file, and another file, at the same directory level (not in a `www/` folder), which will be HTML and which you can name anything you like. Let's call ours `template.html`.

# Inline template code

Let's look first at the slightly simpler method, which is including the code inline within the HTML template. We'll take some elements of the Gapminder application in the previous chapter and incorporate them into an HTML template. The `server.R` file is unchanged, although this version is a little shorter than the previous one because we have taken out some of the outputs. For the sake of brevity, we'll again load the data that we generated in the previous chapter. If you want to see how to generate this data, have a look back at the previous chapter.

# server.R

To remind you of the `server.R` file, and to show you which bits we have retained, let's look at the `server.R` file first, shown in the following code:

```
library(tidyverse)
library(gapminder)

load("geocodedData.Rdata")

function(input, output) {

 theData = reactive({
 mapData %>%
 filter(year >= input$year[1])
 })

 output$trend = renderPlot({
 thePlot = theData() %>%
 group_by(continent, year) %>%
 summarise(meanLife = mean(lifeExp)) %>%
 ggplot(aes(x = year, y = meanLife, group = continent,
 colour = continent)) + geom_line() +
 ggtitle("Graph to show life expectancy by continent over time")

 if(input$linear){
 thePlot = thePlot + geom_smooth(method = "lm")
 }

 print(thePlot)
 })
}
```

There are no surprises in this code, really; it is just a cut-down version of the code from the previous chapter. You can see a reactive function to bring back the data (`theData()`), and a call to `renderPlot()`, which produces a line graph using `ggplot()`.

# ui.R and template.html

The `ui.R` is incredibly simple, and looks like this:

```
| htmlTemplate("template.html")
```

That's it! You are just telling Shiny what the HTML file is called. Remember that we called it `template.html`. Simple. Now, the real work is done with the HTML file.

Let's look at the HTML file's various sections. First, the `head`:

```
| <html>
| <head>
| {{ headContent() }}
| {{ bootstrapLib() }}
| </head>
```

The first thing to note is how the file will be written. Essentially, it will be HTML, interspersed with Shiny code wrapped in double curly braces. So, this first section opens the `html` tag and the `head` tag and then runs two functions, which are given inside double curly braces. You always need to include `headContent()`, which places the boilerplate code necessary to make the page work, just as we saw with the minimal HTML web page that we looked at earlier in the chapter. Some Shiny functions require you to include `bootstrapLib()`, which loads various Bootstrap components. It's not really worth worrying about whether you are using those components or not, since you might add them later, or include them at first and then take them away, and so on. Just include it every time and then you won't need to worry.

Next, we define the body of the application, as shown in the following code:

```
| <body>
| <h1>Minimal HTML UI</h1>
|
| <div class="container-fluid">
| <div class="row">
| <div class="col-sm-4">
| <h3>Control panel</h3>
|
| {{ sliderInput(inputId = "year",
| label = "Years included",
| min = 1952,
| max = 2007,
| value = c(1952, 2007),
```

```

 sep = "",
 step = 5) }]

{{ checkboxInput("linear", label = "Add trend line?",
 value = FALSE) }}

```

```

</div>

<div class="col-sm-8">

{{ plotOutput("trend") }}

<p>For more information about Shiny look at the
documentation.</p>

<hr>

<p>If you wish to write some code you may like to use the pre() function like thi
<pre>checkboxInput("linear", label = "Add trend line?", value = FALSE)</pre>

```

```

</div>
</div>
</body>
</html>

```

As you can see, the Shiny code is included just as you would see it in any normal Shiny application (indeed, these are the functions from [Chapter 2, Shiny First Steps](#)), but in each case, they are wrapped with double curly braces. It's as simple as that. It is possible to pass values into the template from the `ui.R`, like this:

```
| htmlTemplate("template.html", customStep = 10)
```

This value can now be accessed from within the Shiny code in the template, like so:

```

{{ sliderInput(inputId = "year",
 label = "Years included",
 min = 1952,
 max = 2007,
 value = c(1952, 2007),
 sep = "",
 step = customStep) }}

```

# Defining code in the ui.R file

The other method of defining an HTML template is very similar, except in this case, the Shiny code is defined in the `ui.R` file. The `server.R` code is unchanged.

# ui.R

Let's first look at the `ui.R` file, shown in the following code:

```
htmlTemplate(
 "template.html",
 slider = sliderInput(inputId = "year",
 label = "Years included",
 min = 1952,
 max = 2007,
 value = c(1952, 2007),
 sep = "",
 step = 5),
 checkbox = checkboxInput("linear", label = "Add trend line?", value = FALSE),
 thePlot = plotOutput("trend")
)
```

As you can see, in this case, we are defining the inputs in the `ui.R` file. This is just like passing the value of `customStep` in the previous example, except that in this case we are passing whole widgets, not just values. These values are now referred to very simply in the `template.html`, like so:

```
... preamble

<div class="container-fluid">
 <div class="row">
 <div class="col-sm-4">
 <h3>Control panel</h3>

 {{ slider }}

 {{ checkbox }}
 </div>

 <div class="col-sm-8">

 {{ thePlot }}

 <p>For more information about Shiny look at the
 documentation.</p>

 <hr>
 ... rest of HTML
```

The methods are very similar. Defining inline is more appropriate when your Shiny code is relatively brief, as it was in this example. If you have large amounts of Shiny UI code, you will probably prefer to define it in `ui.R` to avoid cluttering your HTML file with R code.

# **Take a step back and rewind**

We covered quite a lot of material in this section, so it's worth spending a moment reviewing what we covered. We've looked at adding HTML to an application that is otherwise written in pure Shiny. We've looked at styling applications with CSS, as well as building entire interfaces with HTML using HTML templates, or by making the whole thing from scratch.

# 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 UI design, and though you are welcome to plow 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 9, Persistent Storage and Sharing Shiny Applications](#).

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 with 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 that 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!

# Debugging

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. The `cat(str(x))` phrase is particularly useful, and will print a little 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 debug mode (see `?browser`). There is more on debugging Shiny applications in [Chapter 8, \*Code Patterns in Shiny Applications\*](#).

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.

# Bootstrap 3 and Shiny

Since the first edition of this book, Shiny has migrated from Bootstrap 2 to Bootstrap 3. There are now many functions—both within Shiny and in its packages (such as `shinydashboard` and `shinythemes`, both available on CRAN)—that enhance the way you can style your applications straight from Shiny. There is more on advanced layout functions, Bootstrap 3, and the packages to help you to style your applications in [Chapter 4, \*Mastering Shiny's UI Functions\*](#).

# Summary

This chapter has incorporated quite a pile of tools into your Shiny toolbox. You 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.

In the next chapter, you are going to learn more about building a UI with Shiny. Shiny includes a lot of different layout functions, which will all be described, and we will also look at making nice tables, using progress bars, and modals, as well as making your UI reactive.

# Mastering Shiny's UI Functions

So far in this book, we've mastered the basics of Shiny by building our own **Gapminder** data-explorer application, and we've looked at how to style and extend Shiny applications using HTML and CSS. In this chapter, we are going to extend our toolkit by learning about more of Shiny's UI functions. These allow you to take control of the fine details of the way your application looks and behaves.

In order to do this, we're going to change the way the UI works in the Gapminder application to make it more intuitive and well-featured. The finished code and data for this advanced Gapminder application can be found at <https://chrisbeeley.net/website/>.

In this chapter, we will cover the following topics:

- Using Shiny's many layout functions effectively
- Learning how to show and hide parts of the interface
- Changing the interface reactively
- Producing beautiful tables with the DataTables library
- Showing progress bars to users in long-running functions
- Showing messages with modals

# Shiny's layout functions

There are quite a lot of different layout functions in Shiny, and understanding what they all do can help you to build the interface that you want easily. It's possible to combine a lot of them, too, so we'll review the form and function of each, and then show a larger example at the end that combines them together. There are three main types of layout function that Shiny offers. The first is what I would call **simple** layout functions. These produce a straightforward kind of layout, without much in the way of styling, and it is these functions that can often be combined together. The next kind is what I would call **complete**. This is a layout function that offers a little styling and is suitable for defining an entire application. It would therefore often be used on its own without any other type of layout function. The last kind I would call the **do-it-yourself** and it refers to a function that doesn't really do much at all, but rather just defines the space in which you will place your widgets. This function, of course, is very suitable for combining with other functions to produce the setup you want. Let's review each type in turn.

# Simple

Throughout these code examples, we're going to have a very simple `server.R` file, which will not change. It simply draws a table and nothing else. We will therefore also break the habit of the rest of this book and use the single `app.R` file structure for these examples, since they are so small. With that in mind, let's look at the first application, which will be `flowLayout()`:

```
server = function(input, output) {
 output$table = renderTable({
 head(iris)
 })
 ui = flowLayout(
 sliderInput("slider", "Slider", min = 1, max = 100, value = 50),
 textInput("text", "Text"),
 tableOutput("table")
)
 shinyApp(ui, server)
```

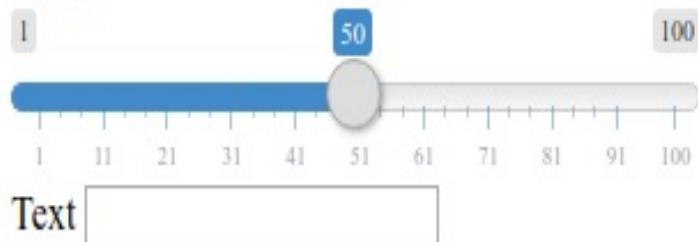
In `flowLayout`, elements are ordered left to right, top to bottom. Resizing the window causes the elements to reorder themselves so they fit, left to right, top to bottom. Download the code example and try it for yourself. Throughout this section, it's a good idea to download the examples and try them out.

The next example is `verticalLayout()`. In this and subsequent code examples, we will not bother with the `server` and `shinyApp()` code since they will never change. We will just look at the value for `ui` in each case:

```
ui = verticalLayout(
 sliderInput("slider", "Slider", min = 1, max = 100, value = 50),
 textInput("text", "Text"),
 tableOutput("table")
)
```

As the name suggests, it merely arranges elements vertically. As many as you supply. Here it is in action:

## Slider



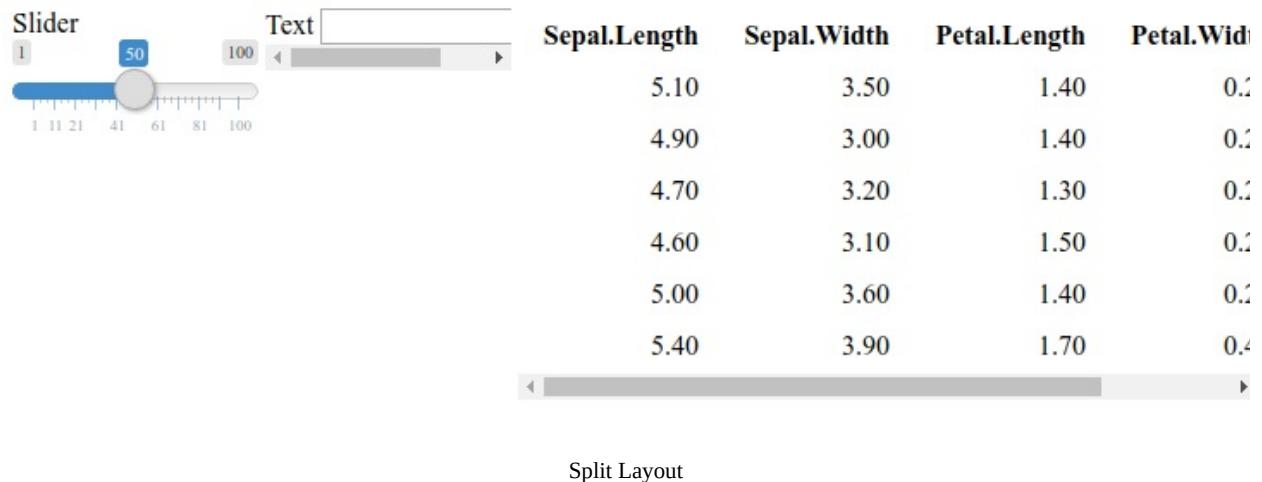
Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.10	3.50	1.40	0.20	setosa
4.90	3.00	1.40	0.20	setosa
4.70	3.20	1.30	0.20	setosa
4.60	3.10	1.50	0.20	setosa
5.00	3.60	1.40	0.20	setosa
5.40	3.90	1.70	0.40	setosa

The final simple function is `splitLayout()`. It takes as many elements as you give it and arranges them on the page left to right. By default, each is given the same width but you can optionally set the width for each manually. Here is an example:

```
ui = splitLayout(
 cellWidths = c("20%", "20%", "60%"),
 sliderInput("slider", "Slider", min = 1, max = 100, value = 50),
 textInput("text", "Text"),
 tableOutput("table")
)
```

In this case, we have set the widths to better accommodate the table (which, as you can see, is given 60% of the width). It's superficially similar to `flowLayout()`, in that if you lay out an application with both, they may look similar at first, but the two critical differences are that `splitLayout()` allows you to define the width of each widget, and that if you resize the window, `flowLayout` (true to its name) will flow onto the next row, whereas `splitLayout()` will just cramp everything up, as

shown in the following screenshot:



# Complete

Now, we come to the complete setup functions, those that can be used individually to make a well-styled application. First, of course, we have the old favorite, `sidebarLayout()`. Many, many Shiny applications are written using this layout and it is an attractive and functional setup, with a well panel that highlights the controls and a large area for one or more outputs. We've already seen this setup, but as a quick review of the basic structure without all the usual clutter, let's take a look at the `ui` code:

```
ui = fluidPage(
 sidebarLayout(
 sidebarPanel(
 sliderInput("slider", "Slider", min = 1, max = 100, value = 50),
 textInput("text", "Text")),
 mainPanel(tableOutput("table"))
)
)
```

It's worth noting that this layout requires the use of `fluidPage()` to set it up correctly. This function is required for this and another layout function, which we will cover shortly, as well as being useful in its own right (which we will also cover shortly).

The other two complete layout functions are `navbarPage()` and `navlistPanel()`. They work similarly in that both provide buttons for you to page through sets of input and output. Let's look at each. Here is the same set of input and output, put together as a **navbar**:



## Slider



## Text

Navbar

This is one of the selected pages, Inputs. The table is viewable on the Table page, accessed by clicking the Table button on the bar at the top. This is obviously not particularly good UI design, it's just done to illustrate how the functions work.

The code is very simple and just consists of tab panels, like you would find in `tabsetPanel()`:

```
ui = navbarPage("Navbar demo",
 tabPanel("Inputs",
 sliderInput("slider", "Slider",
 min = 1, max = 100, value = 50),
 textInput("text", "Text")),
 tabPanel("Table", tableOutput("table"))
)
```

Navlists work in a pretty similar way, except the pages for the buttons are gathered over on the left, as shown in the following screenshot:

Navlist demo

Inputs

Table

Slider

Text

The code here is very similar, except in this case, as with the `sidebarLayout()` function, a call to `fluidPage()` is necessary:

```
ui = fluidPage(
 navlistPanel("Navlist demo",
 tabPanel("Inputs",
 sliderInput("slider", "Slider",
 min = 1, max = 100, value = 50),
 textInput("text", "Text")),
 tabPanel("Table", tableOutput("table"))
)
```

# Do it yourself

We already saw `fluidPage()` being used to set up the environment for other layout functions to work. But this function can also be used to build a UI from scratch, using the `fluidRow()` function. Using `fluidRow()` within `fluidPage()` allows you to implement the standard bootstrap grid layout, as described at [w3schools.com/bootstrap/bootstrap\\_grid\\_system.asp](https://www.w3schools.com/bootstrap/bootstrap_grid_system.asp). Essentially, the interface is built up row by row using the `fluidRow()` function. Each row can be divided into sections of arbitrary width using the `column()` function. The `column` function takes a numeric argument specifying how wide it should be. The total of all of these arguments in a given row should always be 12. So, for example, a row might consist of columns of width 3 and 9, and the following row, perhaps widths of 4, 4, and 4. In this case, a very simple implementation using `fluidRow()` might look like this:

```
ui = fluidPage(
 fluidRow(
 column(width = 4,
 sliderInput("slider", "Slider", min = 1, max = 100, value = 50),
 textInput("text", "Text")),
 column(width = 8,
 tableOutput("table"))
)
)
```

# Combining layout functions

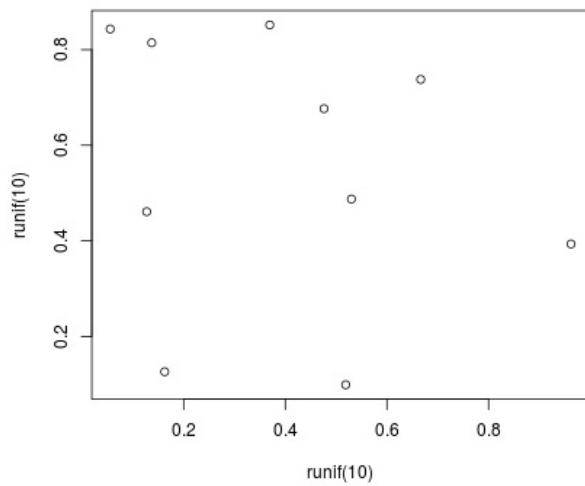
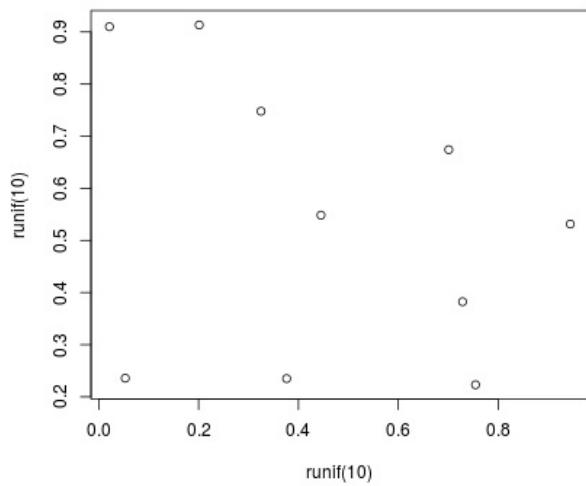
To finish, let's look at how to use different layout functions together to achieve a simple and flexible setup for your application. There are lots of different ways of doing this; this example is designed just to show you that you can use a wide variety of functions to write quick, simple code. In truth, any design can be made using the grid system, given enough time and effort, but it is sometimes simpler just to use a built-in function, as you can see. In this example, I've added a couple of graphs to flesh out the interface a bit, but we won't bother looking at the code for them. They're just created using `renderPlot({plot(runif(10, runif(10)))})` and assigned to `output$graph1` and `output$graph2`. The final UI code looks like this:

```
ui = fluidPage(
 fluidRow(
 column(width = 4,
 sliderInput("slider", "Slider", min = 1, max = 100, value = 50)),
 column(width = 8,
 tableOutput("table"))
),
 splitLayout(
 plotOutput("graph1"), plotOutput("graph2")
),
 verticalLayout(
 textInput("text", "Text"),
 p("More details here"),
 a(href = "https://shiny.rstudio.com/tutorial/", "Shiny documentation")
)
)
```

As you can see, you can freely mix together these layout functions to take advantage of the benefits of each; `fluidRow()` to give precise control over the width of the columns, `splitLayout()` for simplicity, and `verticalLayout()` to stack the widgets on top of each other. The final application looks like the following screenshot:

**Slider**

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.10	3.50	1.40	0.20	setosa
4.90	3.00	1.40	0.20	setosa
4.70	3.20	1.30	0.20	setosa
4.60	3.10	1.50	0.20	setosa
5.00	3.60	1.40	0.20	setosa
5.40	3.90	1.70	0.40	setosa



**Text**

[More details here](#)

[Shiny documentation](#)

# Streamlining the UI by hiding elements

This is a simple function that you are certainly going to need if you build even a moderately complex application. Those of you who have been doing extra credit exercises and/or experimenting with your own applications will probably have already wished for this or, indeed, have already found it.

`conditionalPanel()` allows you to show or hide UI elements based on other selections within the UI. The function takes a condition (in JavaScript, but the form and syntax will be familiar from many languages) and a UI element, and displays the UI only when the condition is `true`. We're going to enhance the Gapminder application to show the option to add a trend line to the graph only when the graph is shown. In order to do this, we're going to have to know which of the tab panels is currently selected, and in order to do that we're going to have to give them a name. Let's do that now.

# Naming tabPanel elements

In order to allow testing for which tab is currently selected, we're going to have to first give the tabs of the tabbed output names. This is done as follows (with the new code in bold):

```
tabsetPanel(id = "theTabs",
 tabPanel("Summary", textOutput("summary"),
 value = "summary"),
 tabPanel("Trend", plotOutput("trend"),
 value = "trend"),
 tabPanel("Map", leafletOutput("map"),
 p("Map data is from the most recent year in the selected range"),
 value = "map")
)
```

As you can see, the whole panel is given an ID (`theTabs`) and then each `tabPanel` is also given a name (`summary`, `trend`, `map`). They are referred to in the `server.R` file very simply as `input$theTabs`.

Finally, we can make our changes to `ui.R` to remove parts of the UI based on tab selection:

```
conditionalPanel(
 condition = "input.theTabs == 'trend'",
 checkboxInput("linear", label = "Add trend line?",
 value = FALSE)
),
```

As you can see, the condition appears very R/Shiny-like, except with the `.` operator familiar to JavaScript users in place of `$`. This is a very simple but powerful way of making sure that your UI is not cluttered with irrelevant material.

# Beautiful tables with DataTable

Later versions of Shiny added support to draw tables using the wonderful `DataTable` jQuery library. This will enable your users to search and sort through large tables very easily. To see `DataTable` in action, visit the homepage at <http://datatables.net/> or run the application featured in this chapter.

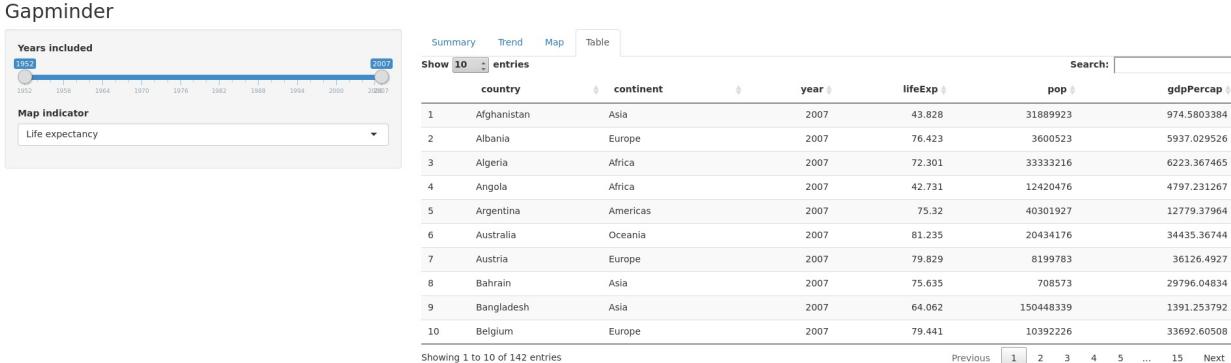
The package can be installed using `install.packages("DT")` and needs to be loaded in the preamble to the `server.R` and `ui.R` file with `library(DT)`. Once this is done, using the package is quite straightforward. There are two functions: one in `server.R` (`renderDataTable`) and one in `ui.R` (`dataTableOutput`). They are used as follows:

```
server.R
output$countryTable = renderDataTable({

 mapData %>%
 filter(year == 2007) %>%
 select(-c(lon, lat))
}

ui.R
tabPanel("Table", dataTableOutput("countryTable"),
 value = "table")
```

Anything that returns a dataframe or a matrix can be used within `renderDataTable()`. Out of the box, it will produce attractive, searchable, pageable tables. In this case, the table looks like this:



The screenshot shows the Gapminder Shiny application interface. On the left, there's a sidebar with 'Years included' from 1952 to 2007, a 'Map indicator' dropdown set to 'Life expectancy', and a color-coded legend for life expectancy. The main area has tabs for 'Summary', 'Trend', 'Map', and 'Table'. The 'Table' tab is active, displaying a DT-based table with columns: country, continent, year, lifeExp, pop, and gdpPercap. The table shows data for 10 countries in 2007. At the bottom, it says 'Showing 1 to 10 of 142 entries' and includes a navigation bar with links for 'Previous', page numbers 1 through 15, and 'Next'.

country	continent	year	lifeExp	pop	gdpPercap
Afghanistan	Asia	2007	43.828	31889923	974.5803384
Albania	Europe	2007	76.423	3600523	5937.029526
Algeria	Africa	2007	72.301	33333216	6223.367465
Angola	Africa	2007	42.731	12420476	4797.231267
Argentina	Americas	2007	75.32	40301927	12779.37964
Australia	Oceania	2007	81.235	20434176	34435.36744
Austria	Europe	2007	79.829	8199783	36126.4927
Bahrain	Asia	2007	75.635	708573	29796.04834
Bangladesh	Asia	2007	64.062	150448339	1391.253792
Belgium	Europe	2007	79.441	10392226	33692.60508

In previous versions of Shiny, there were namespace conflicts between the `datatable` functions and the standard functions, with Shiny drawing a standard

table with some of the same function names as the `DT` package. The situation now is that Shiny will produce these JavaScript data tables out of the box, but without enabling all of the options of the full `DT` package (in particular, client-side processing). It's worthwhile, then, to use the `DT` package and load it in `server.R` and `ui.R`. Let's customize the table using the options that are available with the `DT` package. The code is as follows:

```
datatable(
 mapData %>%
 filter(year == 2007) %>%
 select(-c(lon, lat)),
 colnames = c("Country", "Continent", "Year", "Life expectancy",
 "Population", "GDP per capita"),
 caption = "Country details", filter = "top",
 options = list(
 pageLength = 15,
 lengthMenu = c(10, 20, 50))
)
```

The first thing to notice is that we need to wrap the function in `datatable()`. The `renderDataTable()` function will automatically apply that to any dataframe or matrix returned by it, but in order to add options we need to explicitly add it. You can see the column names being made friendlier with the `colnames` argument, a caption, and a column filter being added at the top using `filter = 'top'`. Lastly, there are some options that are added from within a call to `options = list(...)`, in this case changing the size of the table and the options for row numbers given for the table (making 10, 20, or 50 rows the options available). There are many, many options available, so read the documentation for more details: [rstudio.github.io/DT/](https://rstudio.github.io/DT/).

# Reactive user interfaces

Another trick you will definitely want up your sleeve at some point is a reactive user interface. This enables you to change your UI (for example, the number or content of radio buttons) based on reactive functions.

For example, consider an application that I wrote related to survey responses across a broad range of health services in different areas. The services are related to each other in quite a complex hierarchy, and over time, different areas and services respond (or cease to exist, or merge, or change their name), which means that for each time period the user might be interested in, there would be a totally different set of areas and services. The only sensible solution to this problem is to have the user tell you which area and date range they are interested in and then give them back the correct list of services that have survey responses within that area and date range.

The example we're going to look at is a little simpler than this, just to keep from getting bogged down in too much detail, but the principle is exactly the same and you should not find this idea too difficult to adapt to your own UI. We are going to make a selector, which allows the user to pick one of the years within the year range that they have selected and have that year plotted on the map, instead of being stuck with the most recent year. The choice will be constrained by the fact that the data was only recorded every seven years. So, for example, if the user selects 1960 to 1990, the selector box will contain only the years 1962, 1967, 1972, 1977, 1982, and 1987.

# The reactive user interface example – server.R

When you are making a reactive user interface, the big difference is that instead of writing your UI definition in your `ui.R` file, you place it in `server.R` and wrap it in `renderUI()`. Then, all you do is point to it from your `ui.R` file.

Let's take a look at the relevant bit of the `server.R` file:

```
output$yearSelectorUI = renderUI({
 selectedYears = unique(mapData$year)
 selectInput("yearSelector", "Select year",
 selectedYears)
})
```

The first line takes the reactive dataset that contains only the data between the dates selected by the user and gives all the unique values of the year within it. The second line is a widget type that we have not used yet that generates a combo box. The usual `id` and `label` arguments are given, followed by the values that the combo box can take. This is taken from the variable defined in the first line. The output itself, that is, the bit defined as `output$something = renderUI({...})`, can be given any name you like. It will simply be called by whatever that name is in `ui$output` within `ui.R`, as we shall see in a moment. Note that the actual input that we are creating, that is, the thing that will be called by `input$something`, has the name given in the `selectInput()` function. So, in this case the input will be called `input$yearSelector` and not `input$yearSelectorUI`. To avoid confusion, I have adopted a naming convention for my use of `renderUI`, where I give the `output$` the same name as `selectInput()`, or whichever other function it is, but with UI at the end. This helps me not to mix them up and means I can always remember the other if I know the first one. Let's look at the corresponding entry in the `ui.R` now.

# The reactive user interface example – ui.R

The `ui.R` file merely needs to point to the reactive definition, as shown in the following line of code (just add it to the list of widgets within `sidebarPanel()`):

```
| uiOutput("yearSelectorUI")
```

You can now point to the value of the widget in the usual way, as `input$yearSelector`.

There are more advanced things you can do with a reactive UI using the `insertUI()` and `removeUI()` functions, which allow you to insert arbitrary controls or sets of controls and remove controls, respectively. Although they are quite simple to actually produce, making use of them is quite tricky because of the need to keep track of the names given to each input, as well as which have been added and removed so far. As a consequence, we will not look at an example in this book; see the documentation for a simple example and you may wish to bear in mind that it is possible should you ever need it in a large application.

# Progress bars

It is quite common in Shiny applications, and in analytics generally, to have computations or data-fetches that take a long time. Sometimes, it will be necessary for the user to wait for some time before their output is returned. In cases such as this, it is a good practice to do two things: to inform the user that the server is processing the request and has not simply crashed or otherwise failed, and to give the user some idea of how much time has elapsed since they requested the output and how much time they have remaining to wait.

This is achieved very simply in Shiny using the `withProgress()` function. This function defaults to measuring progress on a scale from 0 to 1 and produces a loading bar at the top of the application with the information from the `message` and `detail` arguments of the loading function.

You can see in the following code that the `withProgress` function is used to wrap a function (in this case, the function that draws the map), with message and detail arguments describing what has happened and an initial value of 0 (`value = 0`, that is, no progress yet):

```
| withProgress(message = 'Please wait',
| detail = 'Drawing map...', value = 0, {
| ... function code...
| })
```

As the code is stepped through, the value of progress can steadily be increased from 0 to 1 (for example, in a `for()` loop) using the following:

```
| incProgress(1/3)
```

The third time this is called, the value of progress will be 1, which indicates that the function has completed (although other values of progress can be selected where necessary; see `?withProgress()`). To summarize, the finished code looks as follows:

```
| withProgress(message = 'Please wait',
| detail = 'Drawing map...', value = 0, {
| ... function code...
| incProgress(1/3)
| ... function code...
| incProgress(1/3)
```

```
| ... function code...
| incProgress(1/3)
| ... function code...
| })
```

It's as simple as that. Again, take a look at the application to see it in action. If you need to give your user more detailed information about how far they are through the processing, the `incProgress()` function also takes message and detail arguments, which means you can change the information being given by `withProgress()` as you step through the function. For example, you may wish to write `incProgress(1/3, detail = "Summarizing data")`, perhaps `incProgress(1/3, message = "Generating graph")`, and so on.

# Progress bar with shinyCSSloaders

The `shinyCSSloaders` package makes it even easier to allow your user to see that an output is loading. It is available on CRAN and so can be installed with this:

```
| install.packages("shinyCSSloaders")
```

Outputs including graphs and tables will now show an animated *busy* icon while they load. The code is as simple as wrapping an output in `withSpinner()`, or even just piping it to `withSpinner()`:

```
| withSpinner(plotOutput("myplot"))
```

Or, you can use the following:

```
| plotOutput("myplot") %>%
| withSpinner()
```

# Modals

Modals are a UI element from Bootstrap and are pop-up messages that can tell your user more about what the application is doing. They can be useful to give a user warnings, or to allow the user to request more information about an output if they wish to know more. You can write a modal very simply, using two functions in the `server.R` file. The `modalDialog` function produces the modal, and the `showModal` function shows it. You can use them together just like this:

```
| showModal(modalDialog(
| title = "Warning",
| "This is a warning"
|))
```

This will give us a simple dialog with a title and a main section:

Warning

---

This is a warning

---

Dismiss

But we can do more interesting things than this with modal dialogs. There are two ways that we can expand their functionality. First, because the `modalDialog()` function will accept any Shiny UI elements, not just text, you can add HTML elements such as horizontal rule, with `hr()`, and even input and output. And second, you can generate the modal dialog using the `modalDialog()` function elsewhere, and merely pass the output to `showModal()`. The application doesn't really do anything sensible, but it's just for illustration. We're going to have a button that launches a modal dialog. The modal dialog will contain an action button and an instruction *not* to press it. If the user does press it, a function elsewhere decides whether we're all doomed or they got away with it this time. Let's have a look at the code. We'll start very quickly with the one line that we add to the `ui.R` file to show the button that makes the modal dialog:

```
| actionButton("showModal", "Launch loyalty test")
```

This is the first time that we've seen an action button. Action buttons are simply HTML controls that start off with a value of 0 and increment by 1 each time they're pressed. One doesn't normally use this value, although sometimes it can be useful to know how many times the button was pressed. Instead, the action button is used because reactive functions can form dependencies on it, meaning they will run when the button is pressed. Including a call to an action button (by just writing `input$showModal` on one line of the code) will cause any reactive function, whether it be an output or otherwise, to rerun when the button is pressed. There is also a special function, `observeEvent({})`, that reacts to events such as button pushes, which we're going to use now. For a detailed description of how to use `observeEvent({})` and its cousin, `observe({})`, see [Chapter 8, Code Patterns in Shiny Applications](#). For now, it's enough to know that `observeEvent(input$showModal, {...})` will run every time someone pushes the show modal button.

Having set up the button to launch it, let's produce the first modal:

```
observeEvent(input$showModal, {
 showModal(modalDialog(
 title = "Loyalty test",
 actionButton("dontPress", "Don't press this")
))
})
```

As you can see, straight away we've made things more interesting by including a Shiny function inside the modal rather than just some text. Now, we're going to set up another modal if someone presses the action button in the first:

```
observeEvent(input$dontPress, {
 showModal(testOutcome(sample(c(TRUE, FALSE), 1)))
})
```

You can see the `showModal()` function that accepts a modal dialog object and the `observeEvent()` function, again reacting to a button push. But this time, we're generating the modal with a function, `testOutcome()`. You can see also that we're passing a value to `testOutcome()`, either `TRUE` or `FALSE`, chosen randomly. This lets the function know whether the user has doomed us all or got away with it this time. Let's look at the function:

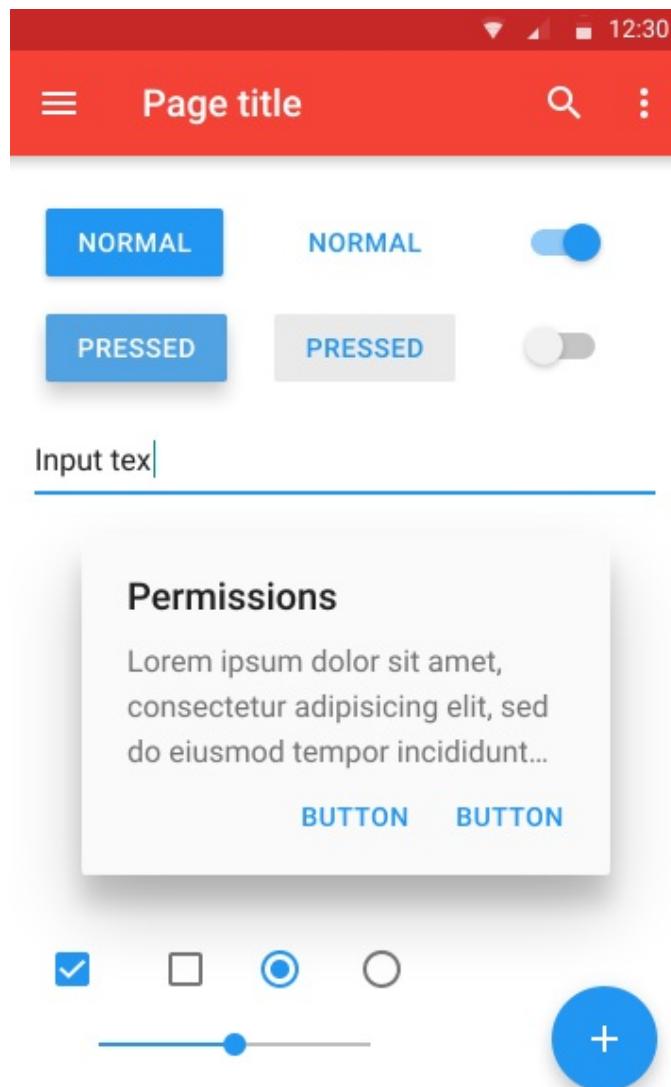
```
testOutcome = function(chance){
 modalDialog(title = "Outcome",
```

```
| ifelse(chance, "You've doomed us all!",
| "You got away with it this time!"))
| }
|
| }
```

As you can see, the function is very simple and creates a modal dialog that decides, based on the value fed to it, whether we're all doomed. Hopefully, that should give you a good grounding in how we can use functions to create modals. If you're comfortable with this, explore the examples in the documentation; they use advanced concepts in Shiny, such as reactive values (covered in [Chapter 8, \*Code Patterns in Shiny Applications\*](#)).

# Alternative Shiny designs

At the time of writing, there is quite an exciting new development in the world of Shiny UI, which is that developers are starting to provide Shiny interfaces to UI frameworks other than Bootstrap. The first example of this, which will hopefully lead to many more, is an implementation of the Material design framework for Shiny. Material design is the very flat design created by Google in 2014 and familiar to any user of the Android operating system. Here is an example:



The Shiny package itself is based on the open source implementation of Material design, Materialize CSS ([materializecss.com/](http://materializecss.com/)), and is called **shinymaterial**. It is available on CRAN ([cran.r-project.org/web/packages/shinymaterial/index.html](http://cran.r-project.org/web/packages/shinymaterial/index.html)). The package itself features different functions from vanilla Shiny, `material_radio_button()` and `material_modal()`, for example, but the principles are the same and any Shiny developer should find it easy to use the different functions to give a totally different feel to their application.

# Summary

In this chapter, we looked at different ways to make your application good-looking and easy to use. We looked at the range of layout functions and how best to use them, and we looked at making input that shows, hides, and changes its contents in response to the state of the application. We also looked at making and customizing data tables, and showing progress bars and messages to your user, as well as a totally new way that Shiny applications can look based on the Material design framework from Google.

The next chapter is all about using JavaScript, all the way from using JavaScript from R code with the `shinyjs` package to producing a complex application that uses JavaScript to pass messages back and forth between the Shiny server and the client.

# Easy JavaScript and Custom JavaScript Functions

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 that fairly minimal amounts of code will be required. In this chapter, we will cover:

- Using JavaScript to read and write to the DOM
- Using JavaScript to send messages between client and server
- Easy JavaScript with the shinyjs package
- Using your own JavaScript with extend Shinyjs
- Listening for events with JavaScript
- Using JavaScript libraries with htmlwidgets

# JavaScript and Shiny

The connection between JavaScript and Shiny is another reason to recommend RStudio as an IDE because it performs beautiful syntax highlighting on JavaScript straight out of the box (although, clearly, other text editors and IDEs may do this or be easily configured to do so).

Before we proceed, it's worth reviewing the difference between server and client-side code and what they're used for. JavaScript gained popularity as a client-side language, which ran on web browsers and added interactivity to websites that would otherwise be static HTML and CSS files, which were downloaded from servers.

It has found increasing use on the server side (for example, with Node.js), but we are going to use it on the client side and so will not consider this any further. So, in this case, JavaScript is running on the client side. The server side in this case, of course, is R, and specifically the commands are to be found in the `server.R` file. Shiny and JavaScript (and, by extension, jQuery) can, as server and client respectively, pass things back and forth between themselves as you wish.

Also, it's worth noting that there are two ways for Shiny and JavaScript to interact with each other.

The first is perhaps the simplest and will be considered first. Because Shiny and JavaScript can both read and write to the web page (that is, to the **Document Object Model (DOM)**), it is quite simple for them to interact with each other on the web page. The DOM is a way of organizing objects in HTML, XHTML, and XML documents, in which elements are referenced within a tree structure.



*A detailed discussion is well outside the scope of this book; suffice to say that if you are going to use JavaScript with Shiny or HTML, you will need to learn about the DOM and how to get and set attributes within it.*

The second way in which Shiny and JavaScript can interact is when it's easier or better to send messages directly between the server and client. Although in theory, you could use the `<input type="hidden">` tag of HTML and pass messages on

the DOM without showing them to the user, it will often be easier to cut out the middle man and send the information directly, particularly when the message is complicated (a large JSON object, for instance).

We will look at sending messages directly after the first example.

First, as a warm-up, we will look at using JavaScript to read the DOM generated by Shiny and perform some client-side processing, before writing the changes back to the DOM.

# Example 1 – reading and writing the DOM

In this example, we're going to find out something about the state of the application from the DOM, grab it with JavaScript, and write it back to the DOM. We'll look at `ui.R` to put together the page and then look at the JavaScript. The `server.R` file is unchanged, so we will not discuss it here.

# ui.R

We're going to use the Gapminder application we've been looking at throughout the book, but add a little JavaScript magic. We're going to add a button that, when clicked, writes the currently selected years on the summary text tab. Let's take a look at what we're adding to the `ui.R` file, over in the `sidebarPanel()` function:

```
| tags$input(type = "button",
| id = "append",
| value = "Add current input values",
| onClick = "buttonClick()"),
|
| includeHTML("appendText.js")
```

As you can see, we use the `tags$xxx()` function that we saw in [Chapter 3, Integrating Shiny with HTML](#), in order to generate a button that will run a JavaScript action when it's clicked. It generates the following HTML:

```
| <input type="button" id="append" value="Add current input values"
| onclick="buttonClick()">
```

The other function, which we also saw in [Chapter 3, Integrating Shiny with HTML](#), is the `includeHTML()` function. As described in [Chapter 3, Integrating Shiny with HTML](#), this function allows you to include HTML from a file rather than cluttering up your `ui.R` with it. Just like `HTML()`, it prevents Shiny from escaping any HTML within it, which is the default behavior. In this case, we are linking to a JavaScript file called `appendText.js`. Now, we just need to add an element to which the JavaScript can write, in `mainPanel()`:

```
| tabPanel("Summary", textOutput("summary"), p(id = "selection", "Values"))
```

We now have a paragraph element with an ID of `selection` to which we can write. `server.R` is unchanged, and so now let's look at the JavaScript.

# appendText.js

The JavaScript is very simple and looks like this:

```
<script type="text/javascript">
 function buttonClick(){
 var elem = document.getElementById('selection');
 elem.innerHTML = document.getElementById('year').value;
 }
</script>
```

You can see that we grab the element with the ID of `selection` and write the value of the year to it. You will, I am sure, wish to produce something a little more sophisticated than this! Now we have the basics, the second example is a bit more complex. In this example, we will be passing messages directly between server (R) and client (JavaScript).

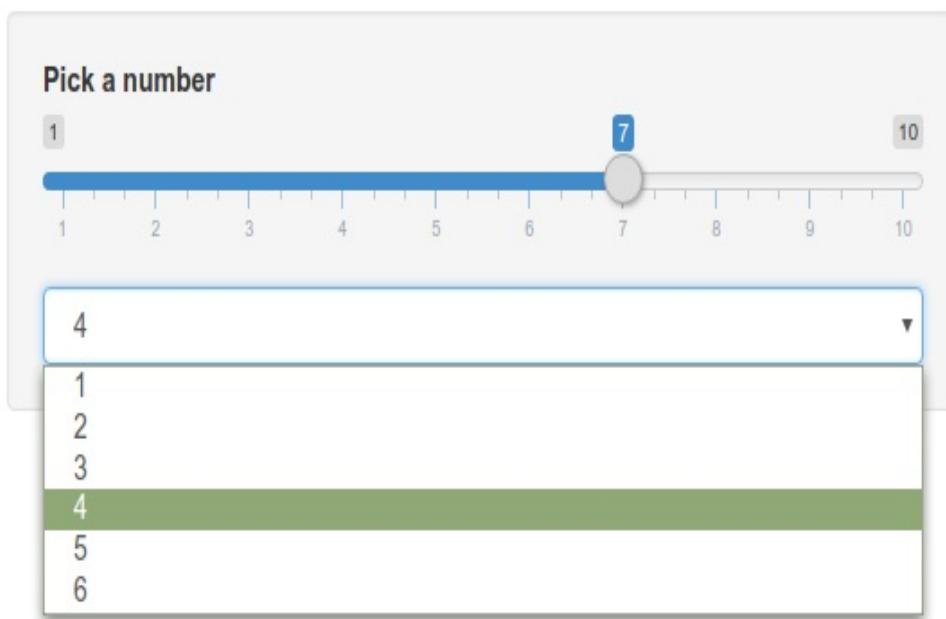
## Example 2 – sending messages between client and server

In this example, we are going to use the DOM and messages to pass information between the client and server. The user will be able to select a number using a slider. The server reads this input and then picks a random number between 1 and the user-supplied number. This number is written back to the screen as well as being sent in a message to JavaScript.

JavaScript receives this number and produces a drop-down selector that allows the user to select a value between 1 and the random number that the server picked. Every time the user selects a different value from this dropdown, JavaScript will decide randomly whether it thinks that Shiny rules! or whether, in fact, JavaScript rules!. This is sent as a message to the server, which picks it up and writes it to the screen. Clearly, this is not of much use as a real application, but it should demonstrate to you the principles of sending and receiving messages and reading and writing to the DOM.

It is definitely worth having a look at the application live. As with all the applications, it can be run straight from my website ([chrisbeeley.net/website](http://chrisbeeley.net/website)) where the source code can also be downloaded. Here is the application in action:

Think of a number:  
Does Shiny or JavaScript rule?



6

---

JavaScript Rules!

As you can see in the previous screenshot, the user has picked 7, the server has picked 6 out of the range of numbers from 1 to 7, JavaScript has built a drop-down menu using that number of options, and has also decided in this case that JavaScript rules. Do note that this application could quite easily be written in pure Shiny, and, like many examples in this book, is provided for illustration only. It is worth keeping it simple, so you can easily see how everything fits together without worrying about understanding everything JavaScript is doing.

In this case, the `ui.R` and `server.R` files are both pretty simple and should be fairly self-explanatory. Most of the code is in the JavaScript file. Let's quickly look at the `ui.R` and `server.R` files first.

# ui.R

The code runs as follows:

```
fluidPage(
 # flexible layout function
 h4(HTML("Think of a number:</br>Does Shiny or </br>JavaScript
 rule?")),
 sidebarLayout(
 sidebarPanel(
 # sidebar configuration
 sliderInput("pickNumber", "Pick a number",
 min = 1, max = 10, value = 5),
 tags$div(id = "output") # tags$XX for holding dropdown
),

 mainPanel(
 includeHTML("dropdownDepend.js"), # include JS file
 textOutput("randomNumber"),
 hr(),
 textOutput("theMessage")
)
)
)
```

The use of `h4(HTML("xxx"))` allows us to shrink the title a little and add some HTML line breaks (avoiding HTML escaping with the `HTML` function as before). `tags$div(...)` produces a `<div>` element in which to place the drop-down menu, which JavaScript will build. The `mainPanel()` call just contains a reference to the JavaScript file that will run on the page, a place to put the random number the server will pick, a horizontal line, and a message from JavaScript regarding whether JavaScript or Shiny rules.

# server.R

The `server.R` file runs as follows:

```
function(input, output, session) {
 output$randomNumber = renderText({
 theNumber = sample(1:input$pickNumber, 1)
 session$sendCustomMessage(type = 'sendMessage',
 message = theNumber)
 return(theNumber)
 })

 output$theMessage = renderText({
 return(input$JsMessage)
 })
}
```

The first thing to note here is the use of `function(input, output, session){...}` instead of the `function(input, output){...}` that we are used to seeing. The addition of a session argument adds a considerable amount of functionality to Shiny applications. In this case, it allows us to send messages to JavaScript. There is more on the functionality of the session argument in the next chapter, [Chapter 6, Dashboards](#).

The first function here carries out two tasks. First, it takes the number that the user selected on the slider and picks a random number between 1 and that number. It sends that number straight to JavaScript using the `session$sendCustomMessage` function (which the session argument we mentioned previously enables). The `sendCustomMessage()` function is defined within Shiny; it is placed after `session$` in order to tie it to the session defined in the `shinyServer(function(input, output, session){...})` function. Finally, it returns the number to Shiny, just like in a standard application, ready to be placed in the output slot, which `ui.R` sets up.

The second function receives the JavaScript message. It's very easy to access, the Shiny function within JavaScript writes it to the standard `input$xxx` variable name, which we are used to seeing throughout the book. As is now plain, a lot of the work in this application is being done within the JavaScript file. Let's take a look.

# dropdownDepend.js

There is quite a lot of code in this section doing quite a lot of different things, so we'll step through each chunk in turn:

```
<script type="text/javascript">
 // Shiny function to receive messages
 Shiny.addCustomMessageHandler("sendMessage",
 function(message) {
 // call this before modifying the DOM
 Shiny.unbindAll();
```

The first part carries out two functions; `Shiny.addCustomMessageHandler("sendMessage", function(message){...})` registers the message-handler with Shiny.

The "sendMessage" name was defined in the `server.R` function in the `the session$sendCustomMessage(type = 'sendMessage', message = theNumber)` call. This is the first step in receiving and processing messages from the server.

The second part begins the process of reading and writing to the DOM. Whenever you are going to modify the DOM in JavaScript, you should call `shiny.unbindAll()` first and then `shiny.bindAll()` at the end; we will come across the latter function later in this section:

```
/* delete the dropdown if it already
exists which it will the second
time this function is called */

// get the dropdown and assign to element
var element = document.getElementById('mySelect');

// if it already exists delete it
if (element !== null) {
 element.parentNode.removeChild(element);
}
```

In this section, we check to see whether the dropdown has already been drawn (which it will be the second time this function is called), and if it has, we delete it in order to redraw it with the new number of options:

```
// Create empty array to store the options
var theNumbers = [];

// add ascending numbers up to the
// value of the message
for (var i = 1; i <= message; i++) {
 theNumbers.push(i);
```

```

 }

 // grab the div ready to write to it
 var theDiv = document.getElementById("output");

```

Now, we create an array and fill it with the numbers from 1 to the value of `message`, which is the random number that the server picked and passed to this function:

```

// create a new dropdown
var selectList = document.createElement("select");

// give it a name and write it to the div
selectList.setAttribute("id", "mySelect");
theDiv.appendChild(selectList);

// add the options
for (var n = 0; n < theNumbers.length; n++) {
 var option = document.createElement("option");
 option.setAttribute("value", theNumbers[n]);
 option.text = theNumbers[n];
 selectList.appendChild(option);
}

```

Next, we create the drop-down list and add the options to it using the array of numbers created immediately before:

```

// add an onchange function to call shinyRules
// every time this input changes
selectList.onchange = shinyRules;

// add class to style nicely in Bootstrap
selectList.className += "form-control";

// call this when you've finished modifying the DOM
Shiny.bindAll();
}
);

```

Finally, we add `onchange` property to the dropdown so that it will call the `shinyRules()` function every time it is changed, add the `form-control` class to render the dropdown nicely in Bootstrap, and call the `shiny.bindAll()` function necessary when we have finished writing the DOM:

```

shinyRules = function(){
 // define text array and pick random element
 var textArray = ['JavaScript Rules!', 'Shiny Rules!'];
 var randomNumber = Math.floor(Math.random()*textArray.length);

 // whenever this input changes send a message to the server
 Shiny.onInputChange("JsMessage", textArray[randomNumber]);
}
</script>

```

This last piece of code defines the `shinyRules()` function, which, as in the preceding code, will be called each time the dropdown is changed. It sets up a text array, picks a random element from it, and then uses the `Shiny.onInputChange`(...) function to send this element to the server. As you can see, the function takes two arguments in this case: `"JsMessage"` and `textArray[randomNumber]`.

The first of these arguments gives the message a name, so it can be picked up by the `server.R` file. This is the part of the `server.R` file that we saw before that reads `input$JsMessage`, so the input is accessed using the standard Shiny notation of `input$xxx` that we are used to seeing. If you go back to look at the `server.R` file, you can see a call to `renderText()` that returns `input$JsMessage`, ready to be written straight to the output panel of the interface.

# Shinyjs

We've already seen that as long as you know JavaScript, using JavaScript is pretty easy in Shiny. The `shinyjs` package, available on CRAN, actually makes it easy to use extra bits of JavaScript in your application by writing pure R code. Install it with `install.packages("shinyjs")` and let's take a look.

Shinyjs gives you access to quite a few nice little JavaScript tricks, so see the documentation for more, but we're going to have a look at a few with the help of the Gapminder application. The first is the ability to disable and enable controls. This can be useful to help your users understand how an application works. For example, in the gapminder application, the year control does not do anything when the map tab is selected, since the data used is always the most recent data anyway. We can hide the control, as we saw in the previous chapter, [Chapter 4, Mastering Shiny's UI Functions](#), but sometimes we may prefer to gray out and disable the control.

In order to use `shinyjs`, we need to call `library(shinyjs)` in both the `server.R` and `ui.R` files. We also need to add `useShinyjs()` anywhere within `fluidPage()` in `ui.R`. With this done, enabling and disabling controls is very simple using the `enable()` and `disable()` functions from Shinyjs. We have named the `tabPanel` `theTabs` and the map tab `map` and therefore we can test whether `input$theTabs` is equal to `"map"`. Now, it's very simple to turn the control on or off:

```
observe({
 if (input$theTabs == "map") {
 disable("year")
 } else {
 enable("year")
 }
})
```

Another simple trick is changing the formatting of text or tables. We can do this very simply using the `toggleClass()` function, which adds and takes away a CSS class to/from a `div`. All we need to do is wrap the text output from the gapminder `ui.R` in `div`, and give it a memorable `id ("theText")`:

```
| div(id = "theText", textOutput("summary"))
```

Define the CSS in the head of the HTML using `tags$head()`:

```
fluidPage(
 tags$head(
 tags$style(HTML(".redText {
 color: red;
 }")
),
 ...)
```

Add a button:

```
| checkboxInput("redText", "Red text?")
```

And now, apply to the named `div ("theText")` the named class ("redText") when the named button (`input$redText`) is pressed:

```
observe({
 toggleClass("theText", "redText", input$redText)
})
```

We can also allow the user to reset some or all of the controls. This can be useful if they cannot remember the defaults and they wish to restore default values for the controls without restarting the application. This function requires only a `div`. We will place a `div` around the year slider to allow the user to restore their defaults easily:

```
div(id = "yearPanel",
 sliderInput("year",
 "Years included",
 min = 1952,
 max = 2007,
 value = c(1952, 2007),
 sep = "")
,)
```

Now, we just need an action button for the user to press:

```
| actionButton("reset", "Reset year")
```

Now, we use `observeEvent()` to listen for the button push, and `reset()` to reset the value of the named `div`:

```
observeEvent(input$reset, {
 reset("yearPanel")
})
```

The last example we're going to show uses the `onevent` function, which will run any piece of R code in response to an event. It will respond to the following events: `click`, `dblclick`, `hover`, `mousedown`, `mouseenter`, `mouseleave`, `mousemove`, `mouseout`, `mouseover`, `mouseup`, `keydown`, `keypress`, and `keyup`. We're going to listen for the `hover` event, which refers to when the mouse pointer is above something. In order to do so, we merely specify the type of event, the ID of the control, and then the R function we wish to run:

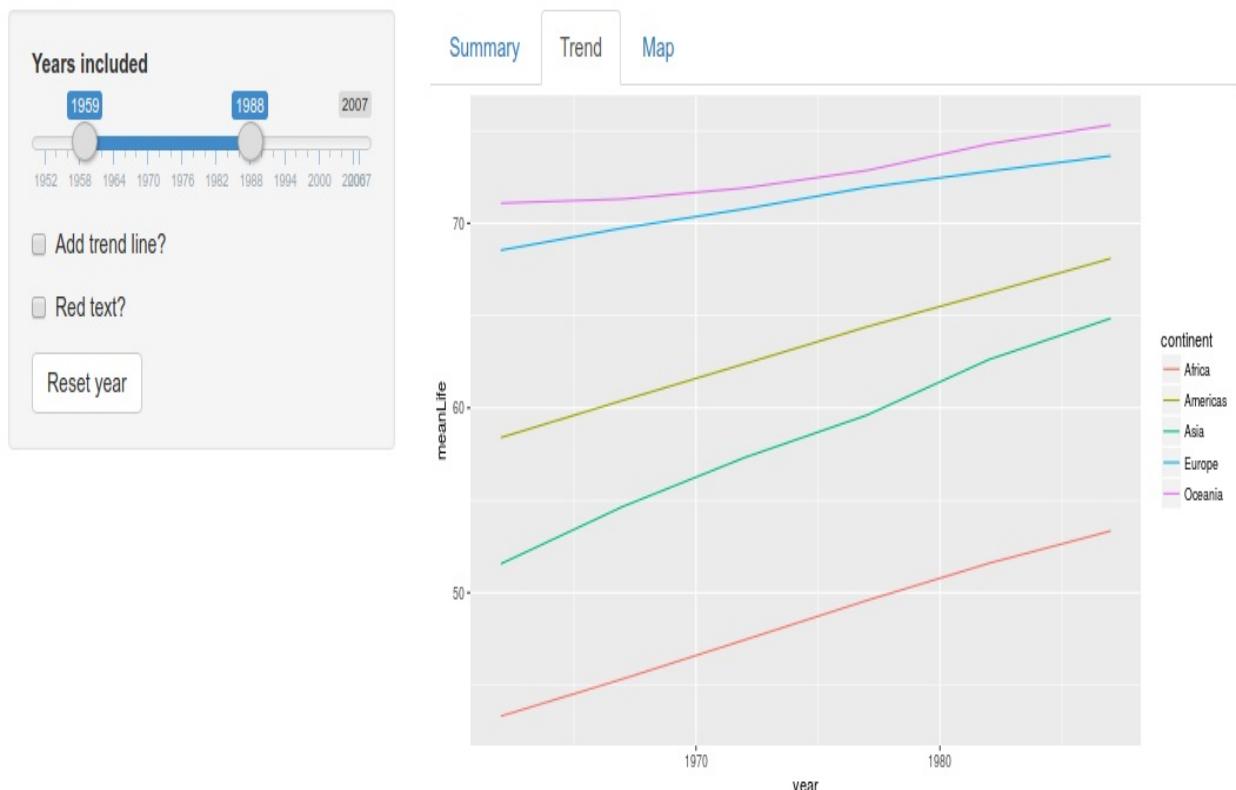
```
| onevent("hover", "year",
| html("controlList",
| input$year, add = FALSE))
```

This function listens for a `hover` over the year control, and then executes the given function. The function looks for an element with an ID of `"controlList"`, and adds the value of the input year. The `add = FALSE` argument is given so the previous entry is overwritten each time. We add the element to the `tabPanel` graph:

```
| tabPanel("Trend", value = "graph", plotOutput("trend"),
| p(id = "controlList")),
```

Now, the application has a little readout of the currently selected years underneath the graph, which updates to the new value whenever you hover over the year control:

# Gapminder



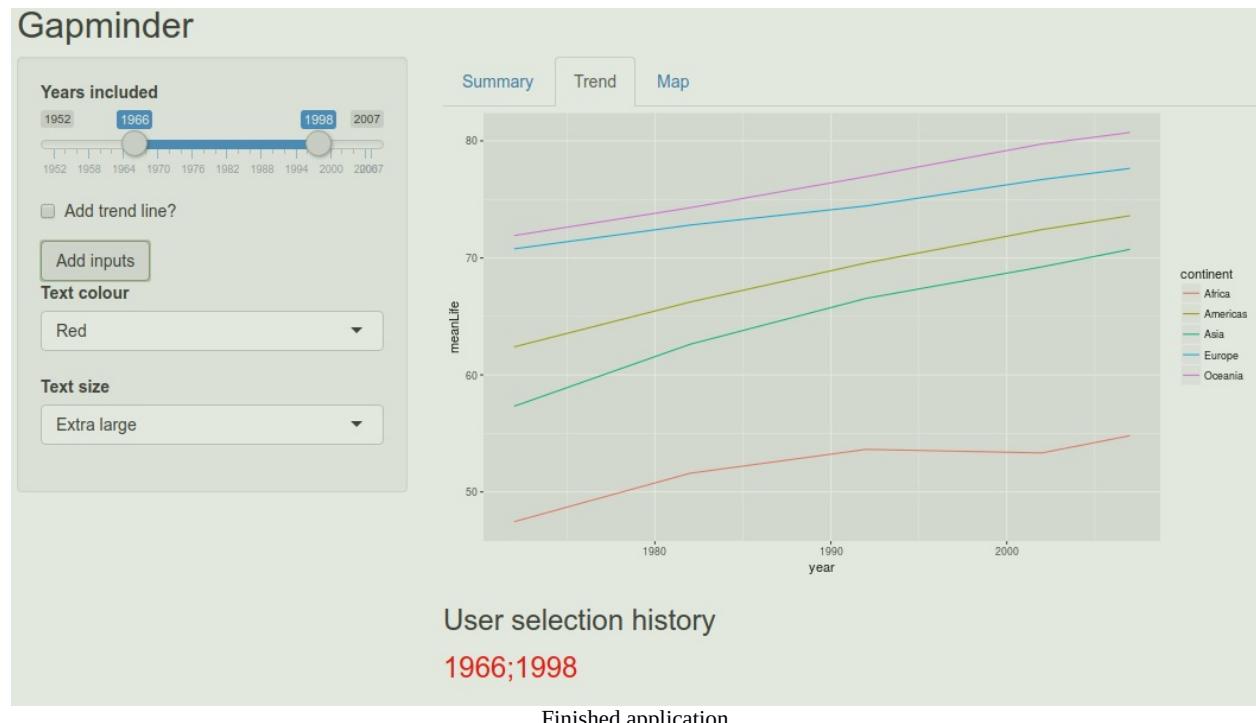
1959, 1988

Gapminder

# Extendshinyjs

The shinyjs package can do much more than just give you access to canned JavaScript, useful though that is. Using the `extendShinyjs()` function, you can very easily read R input and send it to JavaScript. It is simpler to use if the `v8` package is installed. If you cannot install this package, there is a workaround; consult the documentation.

We're going to improve the application that we just produced (it will still be just as useless, of course!) by allowing the user to change the size and the color of the text. The finished application looks as follows:



# ui.R

Let's start with the things to add to the `ui.R`. Once again, we add a place for the text to live to the graph tab:

```
| tabPanel("Trend", plotOutput("trend"),
| h3("User selection history"),
| p(id = "selection", ""))
```

Then, we add a button to add the text and some controls to the sidebar:

```
| actionButton("buttonClick", "Add inputs"),
| selectInput("color", "Text colour",
| c("Red" = "red",
| "Blue" = "blue",
| "Black" = "black")),
| selectInput("size", "Text size",
| c("Extremely small" = "xx-small",
| "Very small" = "x-small",
| "Small" = "small",
| "Medium" = "medium",
| "Large" = "large",
| "Extra large" = "x-large",
| "Super size" = "xx-large"))
```

The last addition is a reference to the JavaScript file that will make everything happen. We add this with the `extendShinyjs()` function. Put it underneath `useShinyjs()`, so you don't lose it:

```
| useShinyjs(),
| extendShinyjs(script = "appendText.js")
```

Let's look now at the `server.R` code.

# server.R

The code here is pretty simple:

```
| observeEvent(input$buttonClick, {
| js$buttonClick(input$color, input$size)
| })
```

We've seen the `observeEvent()` function before; it simply listens for the `input$buttonClick` action button. The `js()` function is doing all the work; it sends its arguments to the `shinyjs.buttonClick` function within the JavaScript file (which we will look at next). In general, `js$foo` sends its arguments to `shinyjs.foo` in the defined JavaScript.

# JavaScript

Lastly, let's look at the JavaScript. As previously mentioned, `shinyjs.buttonClick` is a function that can access the arguments of `js$buttonClick()`. With that in mind, let's look at the code:

```
shinyjs.buttonClick = function(params) {
 // boilerplate code
 var defaultParams = {
 color : "black",
 size : "medium"
 };
 params = shinyjs.getParams(params, defaultParams);

 // rest of code
 var elem = document.getElementById('selection');

 elem.innerHTML = document.getElementById('year').value;
 elem.style.color = params.color;
 elem.style.fontSize = params.size;
}
```

Note that it is not necessary to include the `<script type="text/javascript">...</script>` Script tag around the JavaScript in this file. The first part is boilerplate code, and it's designed to handle the use of named and unnamed lists within the R arguments of the function call. Both are accepted by `extendShinyjs()`, so this code ensures that they play nicely as JavaScript. It also allows you to add default values. You can see the parameters from R being read into the `params` variable with the `shinyjs.getParams` function.

The rest of the code is very simple, merely grabbing the `selection` element, adding text to it, and changing the color and size using `params.color` and `params.size`, both brought through from R and placed as named elements within `params`. You will recall the `input$color` and `input$size` values being passed as arguments within the `js$buttonClick()` function.

And that's it for `shinyjs`. A very simple, powerful way of using either canned JavaScript or incorporating your own JavaScript and reading R input very easily.

# Responding to events in JavaScript

You can listen for all kinds of events in Shiny applications, either from the window or from individual elements, and run JavaScript code when they occur. For example, you can listen for when Shiny makes the initial connection with the client, and you can listen for when Shiny is busy or idle. You can listen for when an input changes, or an output recalculates. For a full list, see the documentation at [shiny.rstudio.com/articles/js-events.html](http://shiny.rstudio.com/articles/js-events.html). As an example, we're going to make a very simple program that takes a long time to draw a graph and gives you a little alert box when it is finished. The program is so simple that we will use the single-file `app.R` format. Here, it is reproduced in its entirety:

```
ui <- fluidPage(
 titlePanel("JavaScript Events"),
 sidebarLayout(
 sidebarPanel(
 actionButton("redraw", "Redraw plot")
),
 mainPanel(
 plotOutput("testPlot"),
 includeHTML("events.js")
)))
server <- function(input, output) {

 output$testPlot <- renderPlot({
 input$redraw
 Sys.sleep(5)
 plot(1:10)
 })
}
shinyApp(ui = ui, server = server)
```

We construct an action button to refresh the graph, set up a very simple plot, and establish a dependency between the plot and the action button so it refreshes when the action button is pressed. The JavaScript file is added using the `includeHTML()` function, as we saw earlier in the chapter.

The JavaScript itself is very simple:

```
<script type="text/javascript">
$(document).on('shiny:idle', function(event) {
 alert('Finished!');
});
</script>
```

You can see that the function listens for `shiny:idle` (which is triggered when Shiny has finished what it is doing) and displays an alert to the user. If we wished, we could just listen for that specific graph being redrawn; this is achieved simply as follows:

```
| $('#testPlot').on('shiny:recalculated', function(event) {
| alert("Finished");
|});
```

You can access a whole range of JavaScript plotting libraries straight from R, and therefore from Shiny, using the `htmlwidgets` package. Let's have a look at some of the things that are available.

# htmlwidgets

The `htmlwidgets` package allows package developers to very easily produce bindings between JavaScript visualization libraries and R. If you wish to make use of the `htmlwidgets` package to produce a binding to your own favorite JavaScript library, it is a relatively simple process, the details of which can be found at [htmlwidgets.org/develop\\_intro.html](http://htmlwidgets.org/develop_intro.html). We will not look at the process of producing your own bindings because many popular libraries are available, and there are plenty in this chapter that demonstrate the use of existing libraries. Moreover, it requires competence with JavaScript, which is not assumed in this book. Suffice to say that the `htmlwidgets` package makes it easy to use JavaScript visualization libraries from R, including R Markdown documents and Shiny applications.

We've already seen `leaflet` in this book. This package makes use of the `htmlwidgets` package. In this chapter, we will have a look at some other useful packages that make use of `htmlwidgets`. For the sake of space, we will look only at the functionality and some example applications, rather than going through all of the code. The application pictured is the one used in the previous edition of this book and all the code for this chapter can be found at [chrisbeeley.net/website/shinybookv2.html](http://chrisbeeley.net/website/shinybookv2.html).

We will look at the following:

- `dygraphs`
- `rCharts`
- `d3heatmap`
- `threejs`

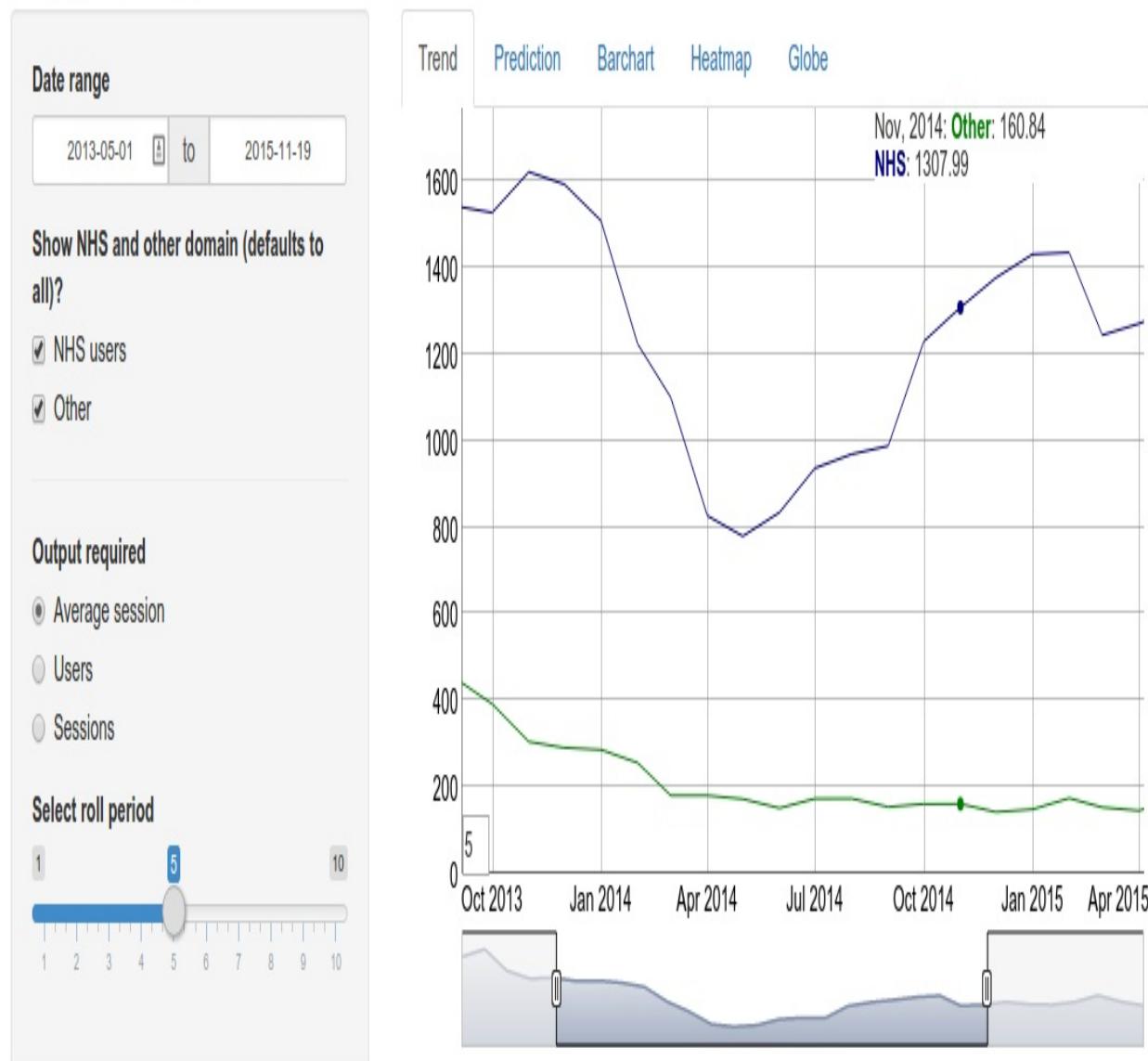
# Dygraphs

The `dygraphs` library in JavaScript (<http://dygraphs.com/>) is designed to show time series and trend data. It supports zoom, pan, and mouseover, and even supports mobile devices by offering pinch to zoom. The `dygraphs` R package provides a handy interface to many of the functions of the `dygraphs` library. It can be installed using `install.packages("dygraphs")`.

For more information about the `dygraphs` package, see the following link, <http://rstudio.github.io/dygraphs/>.

Let's take a look at an example graph:

# Google Analytics



933 days are summarised. There were 24178 users in this time period. 578 days are selected within the plot window

Google Analytics

There are a couple of things you need to make a note of on this graph. First, you can see the mouseover effect at the top-right of the graph, where the date and values of NHS users and Other are listed. Second, this graph has been smoothed

using a rolling average. The number of points to be averaged is specified in the widget on the left-hand side of the page (Select roll period) and is given by default in the small square box at the bottom-left of the graph. Third, the gray box at the bottom with the selector on either side can be used to select date ranges on the graph. This can be useful in a Shiny application as a way of keeping the date range of all the data constants but allowing the user to zoom in on the graph as they choose. As you can see, the text underneath the graph makes this distinction clear, reporting the number of days in the whole dataset (933) as well as the number selected on the graph itself (578). Making a graph user-friendly and interactive is extremely easy using the `dygraphs` package.

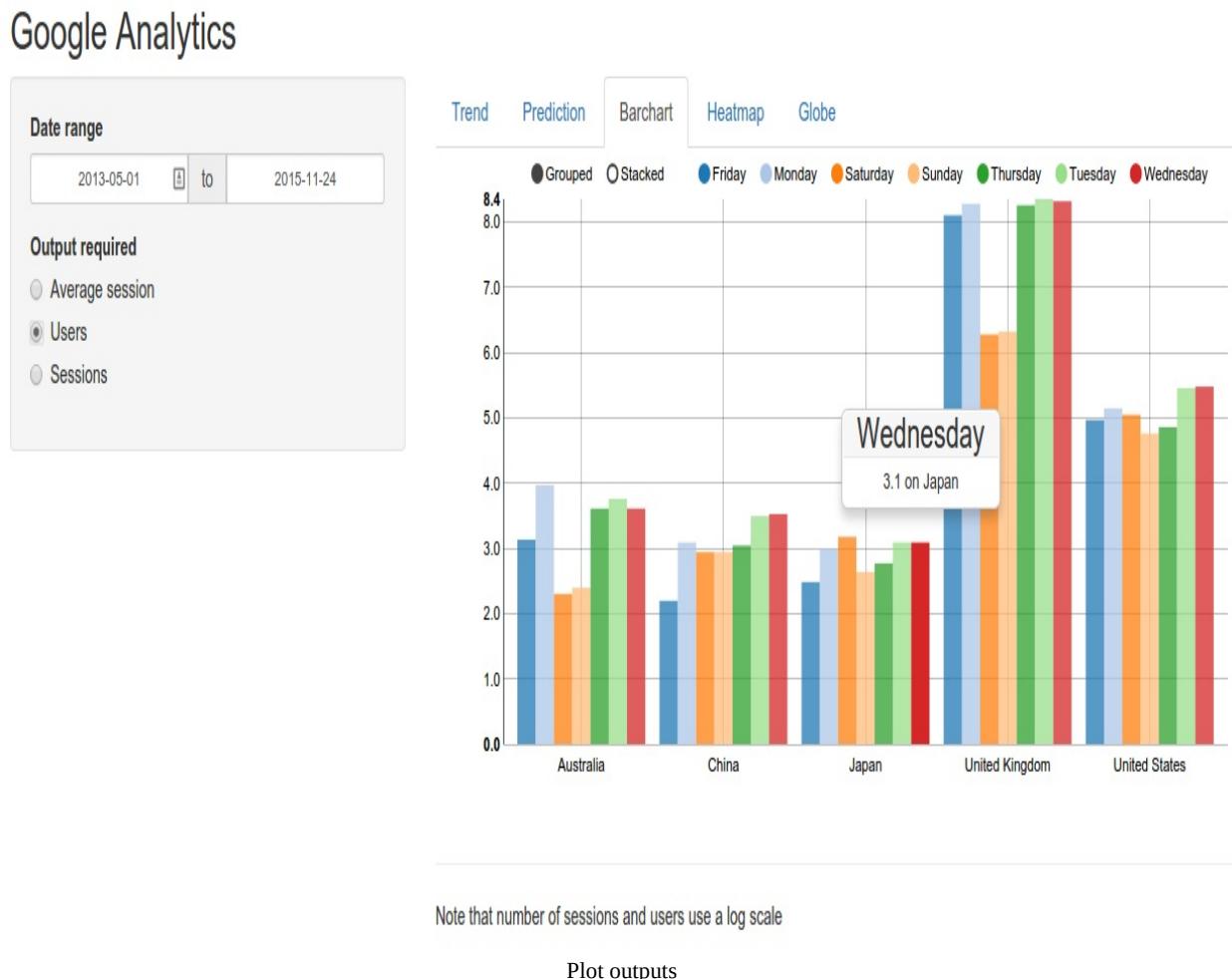
# rCharts

The support in `rcharts` for different JavaScript libraries is very broad, and there are many possibilities for producing beautiful, interactive graphics. Here, we will take a look at just one and peruse the documentation at [ramnathv.github.io/rCharts/](http://ramnathv.github.io/rCharts/) to see the different libraries and graphs supported by the package.

The `rcharts` package is not available on CRAN but can be installed very easily using the following code:

```
| install.packages("devtools") require(devtools) install_github("ramnathv/rCharts")
```

Let's now take a look at one of the many plot outputs possible with this package:



This plot is a clustered bar chart of the selected input (Average session/Users/Sessions), showing the numbers on each day and in five different countries. Thanks to the magic of D3, the plot is interactive out of the box. It supports mouseover, as shown in the screenshot, with Japan's results for Wednesday highlighted. The plot can be changed from Grouped to Stacked (that is, different days grouped horizontally or vertically), and the days of the week can be hidden and shown by clicking on them at the top in the legend of the graph.

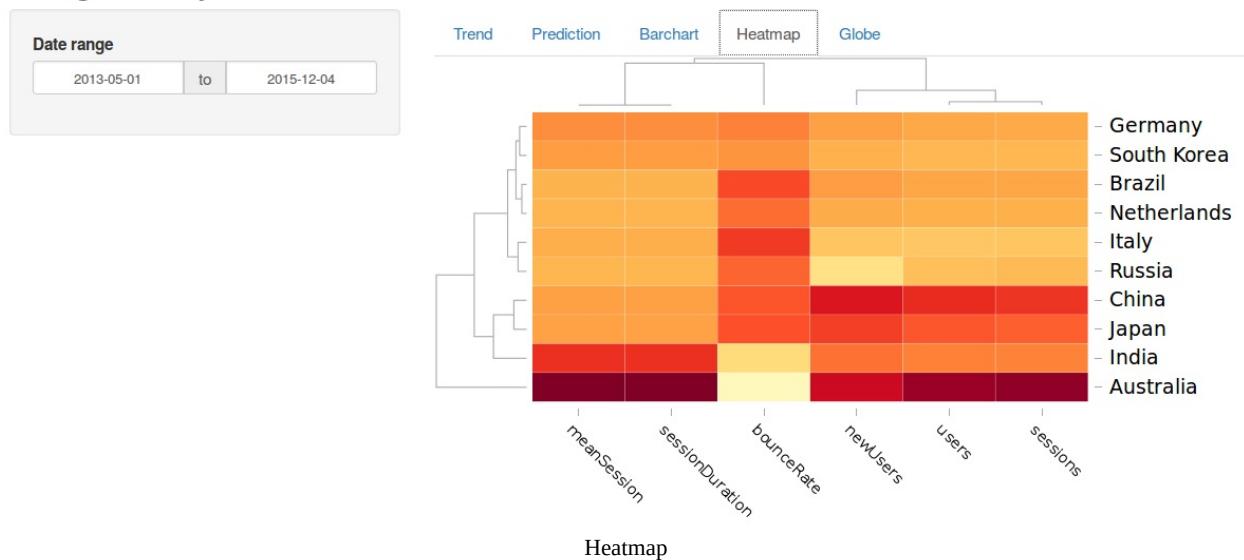
# d3heatmap

The `d3heatmap` package uses vanilla D3 and produces interactive heatmaps. It can be installed using `install.packages("d3heatmap")`.

For more information about the `d3heatmap` package, see the following link, [http://htmlwidgets.org/showcase\\_d3heatmap.html](http://htmlwidgets.org/showcase_d3heatmap.html).

Here is an example:

## Google Analytics

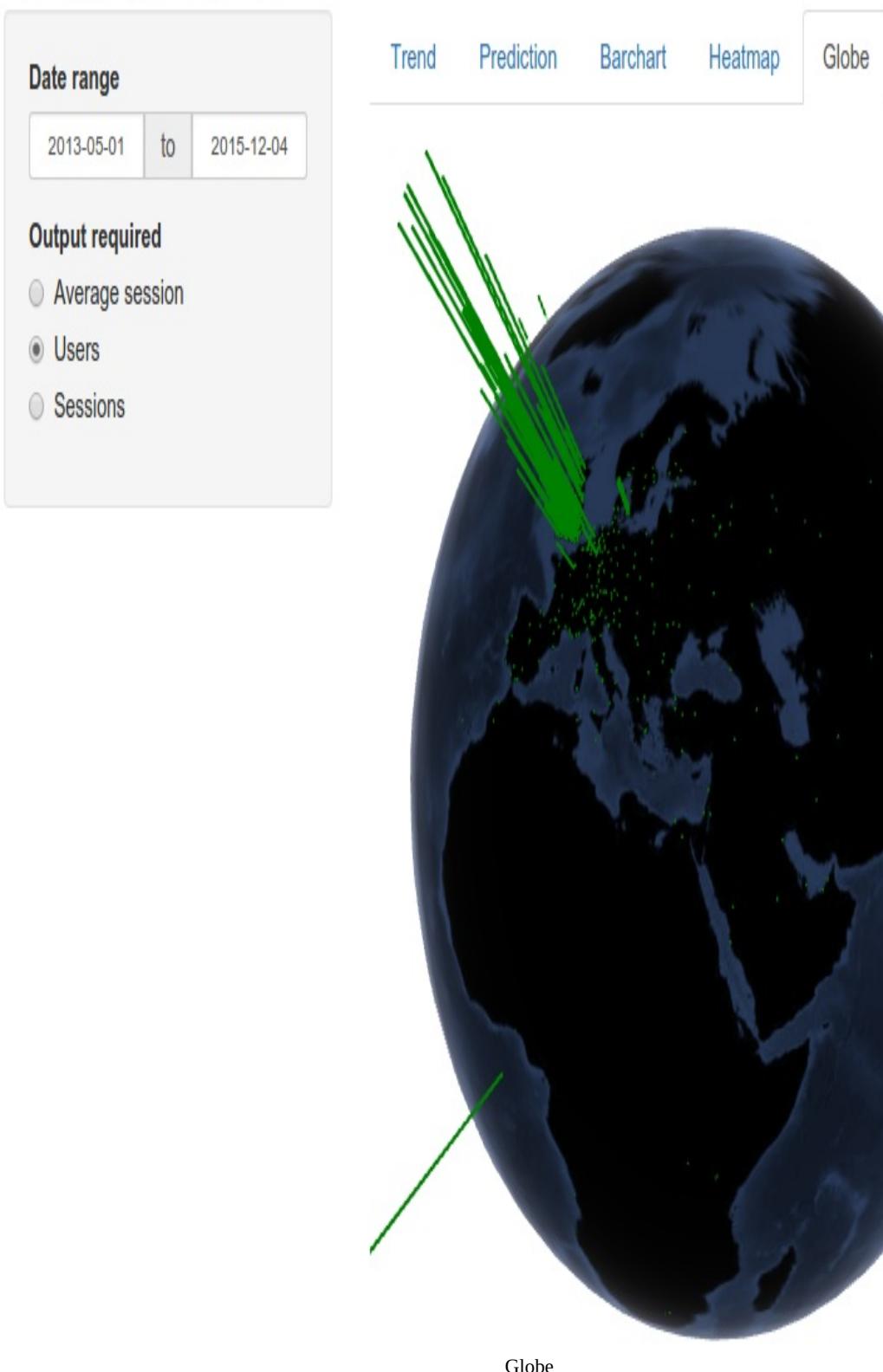


Mouseover gives the individual values within the heatmap.

# threejs

The `threejs` package, as discussed at the beginning of the chapter, can be used to produce 3D scatterplots or plots on globe mappings (including Earth and celestial bodies). It can be installed using `install.packages("threejs")`. In order to produce the globe output that is shown in the screenshot, it is necessary to also install maps using `install.packages("maps")`. An example is shown here:

# Google Analytics



The visualization is completely interactive and can be spun with a mouse drag.

# Summary

In this chapter, we used JavaScript to read and write the DOM, and send messages back and forth with the server. We explored how to get the most out of the `shinyjs` package, as well as how to listen for events with JavaScript. Lastly, we talked about the `htmlwidgets` package, and showed some examples of some of the graphics you can produce using `htmlwidgets`-enabled R packages.

In the next chapter, we will look at how to build dashboards in Shiny.

# Dashboards

This chapter is all about laying out your Shiny applications. In [Chapter 4](#), *Mastering Shiny's UI Functions*, we already looked at doing it by hand, using HTML or CSS, and we already saw how to lay out applications using the Bootstrap grid system. Shiny (and its associated packages) includes loads of functions that allow you to lay out your applications beautifully and simply. This chapter takes the code and applications you have already seen and changes them from the very plain, vanilla-looking layout that the default styling returns to slick, customizable layouts, culminating in a full-featured dashboard.

In this chapter, we will do the following:

- Make a dashboard very easily using the flexdashboard template
- Add icons to applications
- Further explore Shiny using the Bootstrap grid system to lay out applications
- Build a dashboard to help your users access all the information they need from within the same intuitive interface

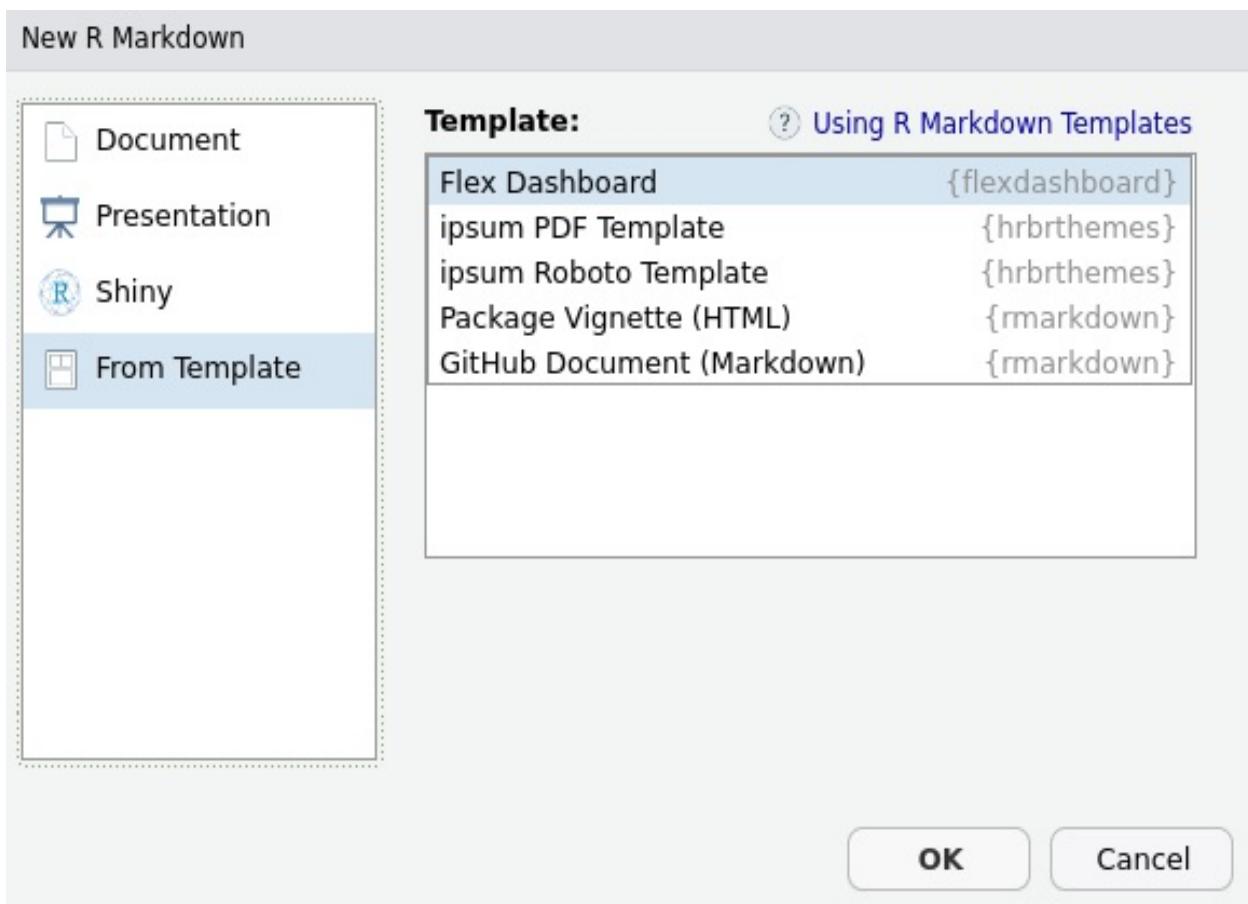
# Applications in this chapter

In order to better understand the layout functions in particular, we're not going to add any functionality in this chapter. We'll start off with the vanilla Gapminder application we have been using throughout and we'll apply different types of layout to it, so you can see how they work. As we progress, we will add one or two extra features. However, we will mainly focus on looking at the same application but with different types of layout functions applied to it.

It is highly recommended that you download and run all the code in this chapter, so you can get a better sense of how the applications work, as well as seeing the `server.R` code in each case, which won't be repeated for each application. If you're not in front of a computer while reading this section, hopefully there are enough screenshots and explanatory material to keep you going until you can see the applications in action for yourself.

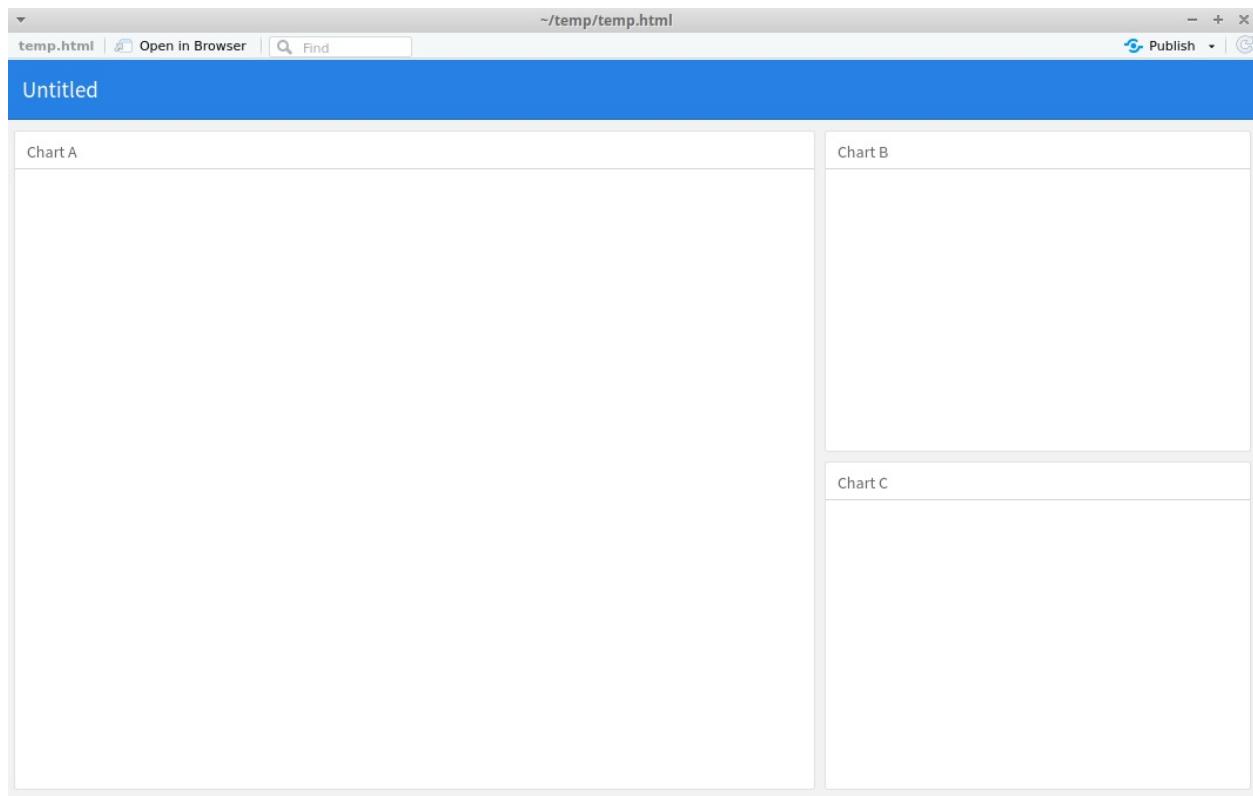
# Flexdashboards

Flexdashboards are a simple R Markdown template and make laying out dashboards (with or without interactivity from Shiny) very simple. For more information about flexdashboards, go to [rmarkdown.rstudio.com/flexdashboard/](https://rmarkdown.rstudio.com/flexdashboard/). Creating a flexdashboard in RStudio is incredibly easy. First, install the package with `install.packages("flexdashboard")`. Then, just select from File | New File | R Markdown... | From Template | Flex Dashboard | OK:



As with most RStudio documents, it comes prefilled with the boilerplate code to make the structure of a dashboard. You can see the structure straight away if you like, by clicking Knit in RStudio (or by calling `rmarkdown::render("yourFileName.Rmd")`).

The boilerplate dashboard looks like this:



Let's make a very simple static dashboard first to learn more about it, and then we'll look at some more advanced layout features and add some interactive Shiny elements. Here is the finished dashboard:



I've used the default layout that is returned when you select a flexdashboard template, which is structured very simply like this, by modifying the headings:

```
Column {data-width=650}

Life expectancy over time
```{r}
*Code*
```

Column {data-width=350}

Life expectancy
```{r}
*Code*
```

GDP per capita
```{r}
*Code*
```

```

As you can see, the dashboard is laid out in columns, with the width defined in {} brackets at each column definition. Chunks are defined as you'd expect in any R Markdown document, with ### as a heading and ```{r} ... ``` wrapping the code. Flexdashboard will neatly arrange the output for you in columns.

Simple! Now, let's look at doing something a bit different with the layout, and

we'll add some Shiny at the same time. To turn a flexdashboard into a Shiny dashboard, just add `runtime: shiny` to the options at the top (this part of the document is called the YAML header). While we're here, let's also lay out the application with rows, rather than columns. Simply change `orientation: columns` to `orientation: rows`.

So, now your header should look like the following:

```

```

```
title: "Shiny gapminder"
runtime: shiny
output:
 flexdashboard::flex_dashboard:
 orientation: rows
 vertical_layout: fill

```

Let's now define a column with a sidebar and place our controls in it as normal, as follows:

```
```{r setup, include=FALSE}
library(flexdashboard)
library(leaflet)
```

Column {.sidebar}

```

```
```{r}
sliderInput("year",
  "Years included",
  min = 1952,
  max = 2007,
  value = c(1952, 2007),
  sep = "",
  step = 5
)
checkboxInput("linear", label = "Add trend line?", value = FALSE)
```

```

And now, we can access those values as normal in the output. One of the output values is defined as normal, except as a row rather than a column:

```
Row

Life expectancy over time
```{r}
renderPlot({
  thePlot = mapData %>%
    filter(year >= input$year[1], year <= input$year[2]) %>%
    group_by(continent, year) %>%
    summarise(meanLife = mean(lifeExp)) %>%

```

```

ggplot(aes(x = year, y = meanLife, group = continent, colour = continent)) +
  geom_line()

  if(input$linear){
    thePlot = thePlot + geom_smooth(method = "lm")
  }

  print(thePlot)
}).

```

Note that, in this case, we can't define a reactive object to filter the year values, as we did before, so we filter by date within each output item (which is bad coding practice in a normal Shiny application, of course). We'll put the other output items in a tabset, as follows:

```

Row {.tabset}
-----
### Life expectancy
```{r}
renderLeaflet({

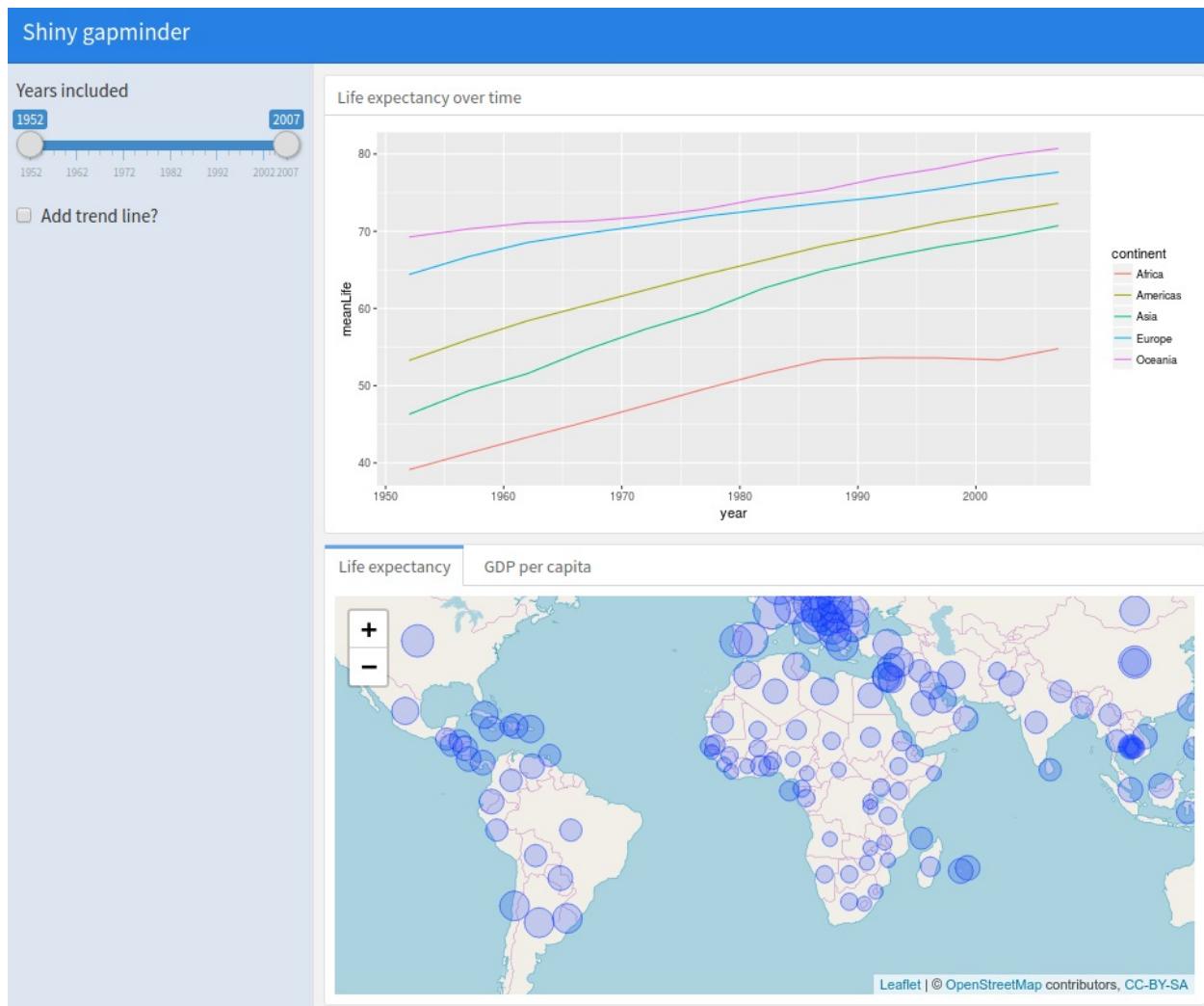
 mapData %>%
 filter(year == input$year[2]) %>%
 leaflet() %>%
 addTiles() %>%
 setView(lng = 0, lat = 0, zoom = 2) %>%
 addCircles(lng = ~ lon, lat = ~ lat, weight = 1,
 radius = ~ lifeExp * 5000,
 popup = ~ paste(country, lifeExp))
}).

GDP per capita
```{r}
renderLeaflet({

  mapData %>%
    filter(year == input$year[2]) %>%
    leaflet() %>%
    addTiles() %>%
    setView(lng = 0, lat = 0, zoom = 2) %>%
    addCircles(lng = ~ lon, lat = ~ lat, weight = 1,
               radius = ~ log(gdpPerCap) * 25000,
               popup = ~ paste(country, gdpPerCap))
}).

```

And that's it. Pretty simple. The finished article looks like this:



It's a pretty smart-looking dashboard, considering how simple it is to code. There are lots more options for flexdashboards, including the use of Shiny modules (covered in [Chapter 8, *Code Patterns in Shiny Applications*](#)) as well as different methods of laying out flexdashboards. Visit the link given previously for more information. For the rest of this chapter, we will be looking at full Shiny applications.

Sidebar application with extra styling

As a warm up, we'll stick to a simple layout with a sidebar for now. As we progress through this chapter, we'll change the layout and make it more like a modern dashboard. Because we're not doing too much in the way of features or layout in this first application, we'll add a few visual bells and whistles. Some of them will be dropped in later versions of the application just to stop the code and UI from becoming too cluttered, but you can of course write an application yourself with all of the UI elements in if you wish to. All of the code and data for the applications in this chapter is available at chrisbeeley.net/website/.

Adding icons to your UI

As we go through the various UI elements later, we're going to sprinkle a few icons throughout, just to give the page a bit more visual interest. Icons can come from two icon libraries, located at fontawesome.io/icons/ and getbootstrap.com/components/#glyphicon. They can be added simply using the `icon()` command with the name of the required icon given as a string.

For example, `icon("user")` will by default return icons from the `Font Awesome` library, and to use the `glyphicon`s, simply add `lib = "glyphicon"` as follows:

```
| icon = icon("user", lib = "glyphicon")
```

They can be added directly to your UI or on buttons (including the buttons at the top of tab panels). From the full code of this application, you can see that we have replaced the boring horizontal rule, which separated our input widgets, with a spinning Linux penguin (because, woo! Linux!) using `class = "fa-spin"`. The `class` argument comes from the use of CSS classes to vary the characteristics of Font Awesome icons. The examples are given at fontawesome.github.io/Font-Awesome/examples/. You can alter the size of Font Awesome icons with `class = "fa-lg"` (one third larger), `class = "fa-2x"` (twice as large), `class = "fa-3x"`, and up to `class = "fa-5x"`. Putting them together, we get the following:

```
| icon("linux", class = "fa-spin fa-3x")
```

Now, let's look at styling your application very easily using `shinythemes`.

Using shinythemes

Let's give the whole application a lick of paint using the `shinythemes` package. If you haven't already done so, install it with `install.packages("shinythemes")`. The documentation (including a list of the available themes) can be found at rstudio.github.io/shinythemes/. Load the package and pass a theme into `fluidPage()`:

```
library(shinythemes)
fluidPage(
  theme = shinytheme("darkly"),
  ... rest of UI...
```

If you want to choose your theme interactively, instead add `themeSelector()` to your UI definition, and a little interactive chooser will appear on your app. Once you're happy with it, use the previous code format to make that the default choice in your app:

```
library(shinythemes)
fluidPage(
  themeSelector(),
  ... rest of UI...
```

Enjoy your new theme and consult the documentation for the other available themes and the appearance of each.

And here's the finished application, showing the spinning penguin and the user and calendar icons, as well as the theme chooser and one of the themes:

Gapminder

Years included

1952

1958 1964 1970 1976 1982 1988 1994 2000 2006 2007

Summary

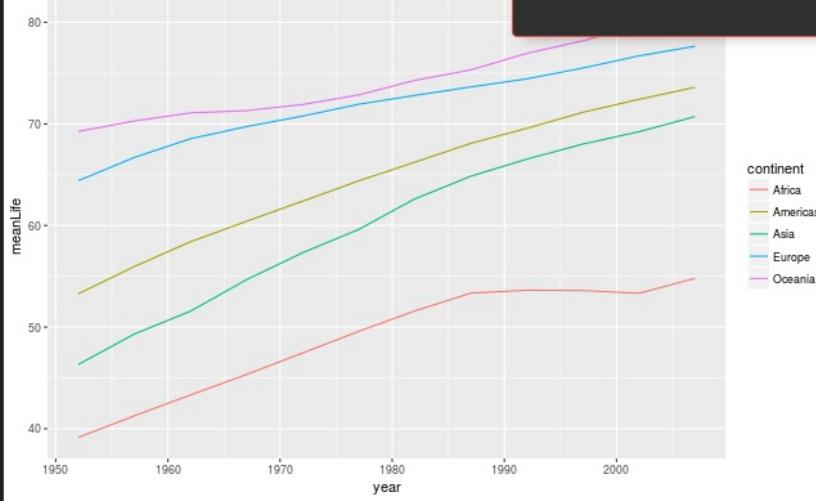
Trend

Select theme:

darkly



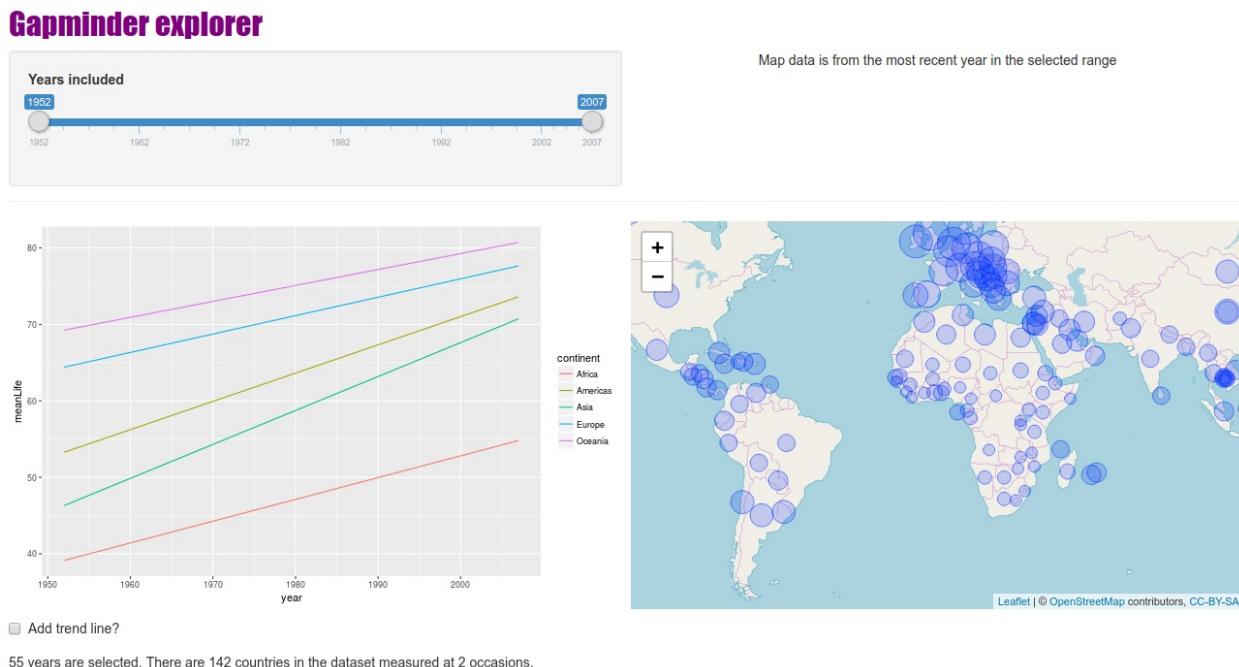
Add trend line?



Using the grid layout

In the next version of the application, we're going to use the `fluidRow()` function to apply a custom layout to the UI. This function allows you to implement the standard Bootstrap grid layout, as we saw in [chapter 4, *Mastering Shiny's UI Functions*](#).

The width of the screen is given as 12 units, and you can pass the `column()` functions of arbitrary size into a `fluidRow()` instruction to define a group of widths adding up to 12. In this simple example, we will have three columns in the first row and then one in the second row. The finished application looks like this:



ui.R

Let's look at the `ui.R` file necessary to achieve this. The `server.R` file remains the same as in the previous example. We'll take breaks as we step through the code to understand what's happening. We'll define the title a little differently here, just to make things a bit different. The actual title itself goes in the `h2()` HMTL helper function, and we use standard inline markup to select the font, color, and size, as shown. The title is given as an argument to the `fluidPage()` function; this gives the browser window its title (it's automatically set by `titlePanel()`, but we're not using that here):

```
fluidPage(  
  title = "Gapminder",  
  
  h2("Gapminder explorer",  
    style = "font-family: 'Impact'; color: purple; font-size: 32px;"),
```

Now, we add in a row of UI elements using the `fluidRow()` function and define two equal columns within this row. Notice the use of `wellPanel()` around `sliderInput()`. This places the input inside a panel and gives it a better, more defined appearance next to the other UI elements, as well as filling the horizontal width of the column:

```
fluidRow(  
  column(6,  
    wellPanel(  
      sliderInput("year",  
        "Years included",  
        min = 1952,  
        max = 2007,  
        value = c(1952, 2007),  
        sep = "",  
        step = 5  
    ))  
,  
  column(6,  
    p("Map data is from the most recent year in the selected range",  
      style = "text-align: center;"))
```

Next, we add a horizontal rule to break the screen up a bit and the two output items, both themselves placed within `fluidRow()`:

```
hr(),  
fluidRow(  
  
```

```
|   column(6, plotOutput("trend")),
|   column(6, leafletOutput("map"))
| ),
```

And add one more fluid row, to place a checkbox where you can choose if you want a trend line and the textual summary, both placed here so they are underneath the relevant graph:

```
| fluidRow(
|   column(6,
|     checkboxInput("linear", label = "Add trend line?",
|                   value = FALSE),
|     textOutput("summary")
|   )
| )
```

Full dashboard

This will be the most full-featured application in this chapter and has the longest `server.R` file.

Ideally, download the application code available at chrisbeeley.net/website/ and run it on your machine, or visit a hosted version, also on the site. There is quite a bit of new functionality in this application, so it's a good idea to explore it now.

If you can't do this, there follows a brief outline of the new functionality in the application:

- Notifications in the top-right of the interface
- Large friendly icons (information boxes) for key figures with icons (calendar, person, pie chart, Shiny version, and so on)
- A gauge

The finished dashboard looks like this:



We'll start by looking at the code for each of these additions and then move on to look at how the whole UI is put together using the `shinydashboard` package.

Notifications

The ability to create notifications is part of a larger amount of functionality within `shinydashboard`, which allows you to create messages, tasks, and notifications in the header of your dashboard. For more details, visit rstudio.github.io/shinydashboard/structure.html.

In this example, we'll just add notifications. The code is very similar to the other two types of content. Static notifications can be produced with the `notificationItem()` function as follows (with the optional status and color arguments not used here):

```
| notificationItem(text = "3 users today", icon("users"))
```

In order to produce content dynamically, we need to do a bit more work. On the `server.R` side, the code is as follows. It allows the notification content to be rendered dynamically and called in the `ui.R` file with `dropdownMenuOutput("notifications")`:

```
output$notifications <- renderMenu({  
  countries = length(unique(theData()$country))  
  continents = length(unique(theData()$continent))  
  
  notifData = data.frame("number" = c(countries, continents),  
                        "text" = c("countries", "continents"),  
                        "icon"= c("flag", "globe"))  
  
  notifs <- apply(notifData, 1, function(row) {  
    notificationItem(text = paste0(row[["number"]], row[["text"]]),  
                      icon = icon(row[["icon"]]))  
  })  
  dropdownMenu(type = "notifications", .list = notifs)  
})
```

The values need first to be placed within a data frame, as shown. The dynamic element is given by `countries` and `continents`, defined as the number of countries and continents, respectively. Text describing what the value represents and an icon are also added to the data frame. The final `notifs` value is produced using a function that's given as an example in the help files, as you can see it iterate over the rows of the data frame and change the notifications into the correct value. As you can see, the data frame that we produced is processed row-wise with the

appropriate values for text and icon being produced (`text = "142 countries", icon = "flag"`, and `text = "5 continents", icon = "globe"`). Finally, they are given to `dropdownMenu()` as the argument to `.list`. Notice that we also define the type as `type = "notifications"`.

Having done all this, the actual notifications are passed into the header in the `ui.R` file quite simply, like this:

```
| header <- dashboardHeader(title = "Gapminder",
|   dropdownMenuOutput("notifications"))
```

This looks a little different to the usual structure that we have encountered, which is because, unlike standard Shiny interfaces, Shiny dashboards are constructed from three separate sections: `dashboardHeader()`, `dashboardSidebar()`, and `dashboardBody()`. We will look in detail at the construction of a Shiny dashboard in the section on `ui.R` later.

Info boxes

We have already seen how to use icons earlier in this chapter, but `shinydashboard` makes a nice feature of it by expanding and coloring icons to draw attention to key pieces of information. An info box can be drawn statically as follows:

```
| infoBox(width = 3, "Shiny version", "1.1.0",
|   icon = icon("desktop"))
```

As you can see, the width can be set (using the 12 span rule from the standard Bootstrap functions we saw earlier in this chapter) with a title (`Shiny version`) and value (`1.1.0`) (although you may often wish to pass a number). This function is placed within `dashboardBody()` in the `ui.R` file. For more information on the arguments of this function, type `?infoBox` into the console.

Although you may sometimes wish to hardcode info boxes in this way (to show version numbers of an application, as in this case), in the majority of cases, you are going to produce this content dynamically. In this case, you will as always need to do some preparation on `server.R` first. Here is the code for the first info box:

```
output$infoYears <- renderInfoBox({
  infoBox(
    "Years", input$year[2] - input$year[1],
    icon = icon("calendar", lib = "font-awesome"),
    color = "blue",
    fill = ifelse(input$year[2] < 2007,
                  TRUE, FALSE)
  )
})
```

The first icon is the number of days within the specified range. The first argument gives the icon a title, `Years`, and the second gives it a value (the number of years, calculated by subtracting the first year from the second). You can also select the color of the box and whether the right-hand portion (which contains the text, as opposed to the icon) is filled (a solid color) or not.

As you can see, here we are deciding the fill of the value portion of the icon dynamically. When the largest year in the range is lower than `2007`, the icon will be filled. If it isn't, it won't. See `?ifelse` for help with `ifelse()`.

The other dynamic info boxes are set up in the same way, as follows:

```
output$infoLifeExp <- renderInfoBox({  
  infoLifeExp = theData() %>%  
    filter(year == 2007) %>%  
    group_by(continent) %>%  
    filter(continent == input$continent) %>%  
    pull(lifeExp) %>%  
    mean() %>%  
    round(0)  
  
  infoBox(  
    "Life Exp. (2007)", infoLifeExp,  
    icon = icon("user"),  
    color = "purple",  
    fill = ifelse(infoLifeExp > 60,  
                 TRUE, FALSE)  
  )  
})  
  
output$infoGdpPercap <- renderInfoBox({  
  
  infoGDP = theData() %>%  
    filter(year == 2007) %>%  
    group_by(continent) %>%  
    filter(continent == input$continent) %>%  
    pull(gdpPercap) %>%  
    mean() %>%  
    round(0)  
  
  infoBox("GDP per capita",  
    infoGDP,  
    icon = icon("usd"),  
    color = "green",  
    fill = ifelse(infoGDP > 5000,  
                 TRUE, FALSE)  
  )  
})
```

ui.R

The `ui.R` file to display dynamic info boxes is similar to the function we already saw to display static info boxes, except now the function is `infoBoxOutput()`. Putting all four info boxes together, we now get the following:

```
fluidRow(  
  infoBoxOutput(width = 3, "infoYears"),  
  infoBoxOutput(width = 3, "infoLifeExp"),  
  infoBoxOutput(width = 3, "infoGdpPerCap"),  
  infoBox(width = 3, "Shiny version", "1.1.0",  
  icon = icon("desktop")))
```

As elsewhere, in each case `infoBoxOutput()` is given a string (`infoYears`), which refers to the name of the corresponding output element (`output$infoYears`).

Google Charts gauge

The gauge with GDP per capita is from the excellent Google Charts API. More information on this can be found at developers.google.com/chart/. Fortunately for us, there is an R package to interface with Google Charts, so there is no need to get our hands dirty with a different API. The package is on CRAN and can be installed with `install.packages("googleVis")`.

Install and load it now. The code is as follows:

```
output$gauge = renderGvis({  
  infoGDP = theData() %>%  
    filter(year == 2007) %>%  
    group_by(continent) %>%  
    filter(continent == input$continent) %>%  
    pull(gdpPercap) %>%  
    mean() %>%  
    round(0)  
  
  df = data.frame(Label = "GDP", Value = infoGDP)  
  
  gvisGauge(df,  
    options = list(min = 0, max = 50000,  
                  greenFrom = 5000, greenTo = 50000,  
                  yellowFrom = 5000, yellowTo = 25000,  
                  redFrom = 0, redTo = 5000))  
})
```

A data frame is produced, with the first column being the label for the gauge, and the second the value of the gauge. If you require more than one gauge, simply include multiple rows. In this case, we will just use one row. The gauge is drawn very simply by passing the data frame and a list of options, which are fairly self-explanatory, giving the minimum and maximum for the gauge, as well as the limits where the gauge is green, yellow, and red, if desired. The gauge is drawn very simply in `ui.R` using `htmlOutput("gauge")`.

Resizing the Google chart

So far, so simple. However, there is a problem! Google visualization charts, unlike native R visualizations, are not automatically resized when the browser window changes. We're going to fix this problem very simply, using some values that we can extract from the session argument of `function(input, output, session) { ... }`. Shiny makes lots of things available in this variable, and one of the most useful of these is `session$clientData`. This tells you lots of things about your user's browser, such as the pixel ratio, as well as the height and width of individual output elements within the application. For more on the uses of the session argument, see shiny.rstudio.com/reference/shiny/1.0.2/session.html. In our case, all we need to do is establish a dependency on something within this client data, which will change whenever the browser window resizes.

In this case, `output_trend_width` is perfect. This is the width of the line plot showing life expectancy over time. These variables are named, for example, `output$clientData$output_nameofoutput_width`. When the browser is resized, this property will change. We're not really worried about height because there isn't anything to bump against the gauge below it, only to the left and right. The code to draw the gauge therefore becomes as follows:

```
output$gauge <- renderGvis({  
  # dependence on size of plots to detect a resize  
  session$clientData$output_trend_width  
  [... as before ...]  
})
```

Changing the width of the browser window will now redraw the gauge, which will make it the right size again.

ui.R

Having examined all the new elements, we can have a look at how Shiny dashboards are put together. As was briefly mentioned previously, Shiny dashboards are composed of three pieces; `dashboardHeader()`, `dashboardSidebar()`, and `dashboardBody()`. They can be put together like this:

```
| dashboardPage(  
|   dashboardHeader([...]),  
|   dashboardSidebar([...]),  
|   dashboardBody([...])  
)
```

Or, they can be put together like this:

```
| # produce components  
| header <- dashboardHeader([...])  
| sidebar <- dashboardSidebar([...])  
| body <- dashboardBody([...])  
  
| # assemble  
| dashboardPage(header, sidebar, body)
```

Personally, I have a strong preference for the latter format, since it seems to me to make the code simpler and easier to read (not to mention with less indentation), but you may prefer the other way.

We already saw the header part of the application previously; this is reproduced in the following code for convenience. Note also that, unlike in many Shiny applications, it is necessary to load several packages in the `ui.R` file because there are special functions within those packages, which get called with the file (for example, `dashboardHeader()`, `leafletOutput()`, and others). The top of the `ui.R` file therefore looks as follows:

```
| library(shinydashboard)  
| library(leaflet)  
| header = dashboardHeader(title = "Gapminder",  
|   dropdownMenuOutput("notifications"))
```

The sidebar can contain input widgets, as is typical in Shiny applications, but also buttons to select different tabs of the dashboard, each of which can be set up to have different output elements on it. In this case, we have two tabs: the main one contains the graphs and icons we have spent most of this section discussing,

and the map tab containing the interactive leaflet map.

The code is as follows:

```
sidebar <- dashboardSidebar(  
  sidebarMenu(  
    menuItem("Dashboard", tabName = "dashboard",  
            icon = icon("dashboard")),  
    menuItem("Map", icon = icon("globe"), tabName = "map",  
            badgeLabel = "beta", badgeColor = "red"),
```

The first two items are tab buttons that will allow us to present different sets of output elements to users. Each is given a title (`Dashboard` and `Map`), a name (`dashboard` and `maps`), and an icon (`dashboard` and `globe`). The second item can give users extra information about the tab—in this example, showing that the output elements on that tab are still in beta, using the `badgeLabel` and `badgeColor` arguments, giving a red beta in this case.

The rest of the sidebar setup is familiar from previous incarnations of this application, as follows:

```
  menuItem("Map", icon = icon("globe"), tabName = "map",  
          badgeLabel = "beta", badgeColor = "red"),  
  
  sliderInput("year",  
             "Years included",  
             min = 1952,  
             max = 2007,  
             value = c(1952, 2007),  
             sep = "",  
             step = 5  
,  
  
  selectInput("continent", "Select continent",  
             choices = c("Africa", "Americas", "Asia",  
                        "Europe", "Oceania"))
```

I've added a `"continent"` selector, which the info boxes and gauge respond to, showing the average life expectancy and GDP per capita for the selected continent. Finally, `dashboardBody` is set up using a `tabItems(tabItem(), tabItem())` structure:

```
body = dashboardBody(  
  tabItems(  
    tabItem(tabName = "dashboard",  
            fluidRow(  
              infoBoxOutput(width = 3, "infoYears"),  
              infoBoxOutput(width = 3, "infoLifeExp"),  
              infoBoxOutput(width = 3, "infoGdpPerCap"),  
              infoBox(width = 3, "Shiny version", "1.1.0",  
                      icon = icon("desktop"))),
```

```

    fluidRow(
      box(width = 10, plotOutput("trend"),
          checkboxInput("linear",
                        label = "Add trend line?",
                        value = FALSE)),
      box(width = 2, htmlOutput("gauge"))
    ),
  ),
  tabItem(tabName = "map",
          box(width = 12, leafletOutput("map"),
              p("Map data is from the most recent year in the selected range")))
)
)
)

```

Each tab item will be passed into here; in this case, we have two, as we saw in the previous `code-dashboard` and `map`. Now you can put anything you like in it. In this case, we have a fluid row with four info boxes of `width` of 3 (the total `width` being 12, of course). The first three are the dynamic info boxes that we set up in the `server.R` file, and the third is a static version. The code for the static version, too, is featured earlier in this section. We finish the tab item with another fluid row. Note the use of the `box()` function that draws white boxes around the elements of a dashboard.

We finish with the final tab, `map`, which, as can be seen, contains just one box with the map in and a comment about the data in the map.

Summary

In this chapter, we explored many different ways of laying out the same applications. We looked at both static and Shiny-based interactive flexdashboards. Starting with the standard sidebar layout, we looked at adding icons and using the `shinythemes` package to quickly style a vanilla application. We explored the functionality of the `shinydashboard` package, which allows you to produce tabbed output sections; add in tasks, notifications, and messages for your users; show large friendly icons with key information on; and provide an attractive and professional-looking default appearance.

The key to making the most of the material in this chapter, as well as of Shiny generally, is to remember that you can combine the tools that Shiny gives you in a lot of different ways, depending on your needs and your skill set. There's nothing to stop you from using the `shinydashboard` package with some highly customized HTML in one of the tabs if you need to very precisely build a particular kind of interface. Some developers, who have more experience with JavaScript than with R, may prefer to work with the Google Charts API in JavaScript and use Shiny as just a data workhorse, serving data or statistics straight to the JavaScript function. Think about what you need, and there will usually be a couple of ways to achieve it. The solution you pick will be based partly on producing an application that is simple, clean, and easy to maintain and debug.

In the next chapter, we are going to look at getting the most out of Shiny by using custom URL strings, producing interactive plots, adding passwords, downloading and uploading data, and more.

Power Shiny

In this chapter, we are going to learn many powerful features of Shiny. We will start with animating plots. After that, we will discuss how to read client information and get requests in Shiny. Nowadays, telling a story about the data in interactive and reporting ways is in demand. So, we will go through graphics and report-generation, and how to download them using `knitr`. Downloading and uploading is also an interesting part of any application, which we will see using some examples. Bookmarking the state of the app is an add-on to regenerate the output on the app. We will see a demonstration of the fast development of apps using widgets and gadgets. At the end of the chapter, we will look at how to authenticate the app using a password.

In this chapter, we will cover the following topics:

- Animation
- Reading client information and `GET` requests in Shiny
- Custom interfaces from `GET` strings
- Downloading graphics and reports
- Downloadable reports with `knitr`
- Downloading and uploading data
- Bookmarking
- Interactive plots
- Interacting with tables
- Linking interactive widgets
- Shiny gadgets
- Adding a password

Animation

Animation is surprisingly easy. The `sliderInput()` function, which provides an HTML widget that allows us to select a number along a line, has an optional animation function that will increment a variable by a set amount every time a specified unit of time elapses. This allows you to very easily produce a graphic that is animated.

In the following example, we are going to look at the monthly graph and plot a linear trend line through the first 20% of the data (0-20% of the data). Then, we are going to increment the percentage value that selects the portion of the data by 5% and plot a linear through that portion of data (5-25% of the data). Then, increment by 10-30% and plot another line, and so on.

The slider input is set up as follows, with an ID, label, minimum value, maximum value, initial value, step between values, and the animation options, giving the delay in milliseconds and whether the animation should loop:

```
sliderInput("animation", "Trend over time",
  min = 0, max = 80, value = 0, step = 5,
  animate = animationOptions(interval = 1000,
    loop = TRUE))
```

Having set this up, the animated graph code is pretty simple, looking very much like the monthly graph data except with the linear smooth based on a subset of the data instead of the whole dataset. The graph is set up as before and then a subset of the data is produced on which the linear smooth can be based:

```
groupByDate <- group_by(passData(), YearMonth, networkDomain) %>%
  summarise(meanSession = mean(sessionDuration, na.rm = TRUE),
    users = sum(users),
    newUsers = sum(newUsers), sessions = sum(sessions))
groupByDate$date <- as.Date(paste0(groupByDate$YearMonth, "01"),
  format = "%Y%m%d")
smoothData <- groupByDate[groupByDate$date %in%
  quantile(groupByDate$date,
    input$animation / 100,
    type = 1) : quantile(groupByDate$date,
    (input$animation + 20) / 100,
    type = 1),]
```

We won't get too distracted by this code, but essentially, it tests to see which of the whole date range falls in a range defined by percentage quantiles based on the `sliderInput()` values. Take a look at `?quantile` for more information.

Finally, the linear smooth is drawn with an extra data argument to tell `ggplot2` to base the line only on the smaller `smoothData` object and not the whole range:

```
ggplot(groupByDate, aes_string(x = "Date",
  y = input$outputRequired,
  group = "networkDomain",
  colour = "networkDomain")) + geom_line() +
  geom_smooth(data = smoothData,
  method = "lm", colour = "black")
```

Not bad for a few lines of code. We have both `ggplot2` and Shiny to thank for how easy this is.

Reading client information and GET requests in Shiny

Shiny includes some very useful functionality that allows you to read information from a client's web browser, such as information from the URL (including `GET` search requests) and the size of plots in pixels.

All you need to do, as before, is run `shinyServer()` with a session argument. This causes, among other things, an object to be created that holds information about a client's session, named `session$clientData`.

The exact content of this object will depend on what is open on the screen. The following objects will always exist:

```
url_hostname  # hostname, e.g. localhost or chrisbeeley.net
url_pathname = # path, e.g. / or /shiny
url_port =     # port number (8100 for localhost, can optionally
               # change when hosting, see chapter 5)
url_protocol = # highly likely to be http:
url_search =   # the text after the "?" in the URL. In the
               following
               # example this will read "?person=NHS&smooth=yes".
```

Different output types will yield different information. Plots will give the following information, among other return values:

```
output_myplot_height = # in pixels
output_myplot_width =  # in pixels
```

There are many applications to which this information can be put, such as giving different UIs or default settings to users from different domains, or configuring graphs and other output based on their size (for example, for users who are using mobile devices or 32" monitors). We're going to look at perhaps the most obvious and powerful use of client data: the search string.

Custom interfaces from GET strings

In this example, we're going to produce URLs that allow Shiny to configure itself when the user lands on the page to save them from having to set up their preferences each time. We will make use of two variables: one specifies that a user is only interested in data from the NHS network domain and the other specifies that the user wants a smoothing line present on their trend graph. Users who request a smoothing line will also be taken straight to the trendline tab.

As well as the work with the `GET` query, the only extra bit we will need here is a function to change the selected panel from `tabsetPanel()`. This is done, unsurprisingly, using `updateTabsetPanel()`.

Catering to these different needs is very easily done by creating URLs that encode the preferences and giving them to the different users. To simplify the code, we will pretend that, if they are passed at all, the correct number of search terms is always passed in the correct order. This is a reasonable assumption if you write the URLs yourself. In a real-world example, the URLs are most likely going to be generated programmatically from a UI. Correctly parsing them is not too challenging, but it is not really the focus of the discussion here. The following are the two URLs we will give out:

- `feedbacksite.nhs.uk/shiny?person=NHS&smooth=yes`
- `feedbacksite.nhs.uk/shiny?person=other&smooth=no`

As in the previous example, the code is wrapped in `observe()`, and the first portion of the code returns the search terms from the URL as a named list:

```
| observe({  
|   searchString <- parseQueryString(session$clientData$url_search)  
|   ...  
| } )
```

Having done this, we can then check that `searchString` exists (in case other users land from the default URL) and, if it does, we can change the settings accordingly. The `updateTabsetPanel()` command uses a lot of the concepts we already saw when we read the tab that was selected. The function takes a `session` argument, an `inputId` argument (the name of the panel), and a `selected` argument

(the name of the tab):

```
# update inputs according to query string
if(length(searchString) > 0){ # if the searchString exists

  # deal with first query which indicates the audience
  if(searchString[[1]] == "nhs"){ # for NHS users do the following
    updateCheckboxGroupInput(session, "domainShow",
      choices = list("NHS users" = "nhs.uk",
      "Other" = "Other"), selected = c("nhs.uk"))
  }

  # do they want a smooth?
  if(searchString[[2]] == "yes"){
    updateTabsetPanel(session, "theTabs", selected = "trend")
    updateCheckboxInput(session, inputId = "smooth",
      value = TRUE)
  }
}

})
```

This is clearly a very powerful way to make the experience better for your users completely transparently. You may wish to spend a bit of time setting up a web interface in whatever language you like (PHP, JavaScript, and so on) and correctly parsing the URLs that you generate within Shiny. If you need to handle varying lengths and names of lists, you will need a few extra commands:

- `names(theList)`: Gives you the name of each return value
- `length(unlist(theList))`: Tells you how long the list is

Downloading graphics and reports

The option to download graphics and reports can be added easily using `downloadHandler()`. Essentially, `downloadHandler()` has two arguments that both contain functions: one to define the path to which the download should go, and one that defines what is to be downloaded.

The first thing we need to do is take any functions that are used either in the download graphic request or the report and make them reactive functions, which can be called from anywhere rather than instructions to draw a graph within a call to `renderPlot()`. The effect of this, of course, is that we only have one function to write and maintain rather than one inside the download graphic function, one inside the download report function, and so on. This is achieved very simply:

```
trendGraph <- reactive({  
  ... rest of function that was inside renderPlot  
})
```

The graph can now easily be printed within the trend tab:

```
output$trend <- renderPlot({  
  trendGraph()  
})
```

We'll go through the following code from `server.R` step by step:

```
output$downloadData.trend <- downloadHandler(  
  filename <- function() {  
    paste("Trend_plot", Sys.Date(), ".png", sep="")  
  },
```

This is the `filename` function, and as you can see, it produces `filenameTrend_plot_xx_.png` where `xx` is the current date:

```
content <- function(file) {  
  png(file, width = 980, height = 400,  
    units = "px", pointsize = 12,  
    bg = "white", res = NA)  
  trend.plot <- trendGraph()  
  print(trend.plot)  
  dev.off()  
},
```

This is the `content` function, and as you can see, it opens a `png` device (`?png`), calls

a reactive function named `myTrend()`, which draws the graph, prints to the device, and closes with a call to `dev.off()`. You can set up the `trendGraph()` function simply; in this case, it is just like the function that draws the graph itself, except instead of being wrapped in `renderPlot()` to indicate that it is a Shiny output it is just defined as a reactive function.

Finally, the following is given to tell Shiny what type of file to expect:

```
| contentType = 'image/png')
```

Adding the download button to the `ui.R` file is simple; the `downloadButton()` function takes the name of the download handler as defined in `server.R` and a label for the button:

```
| tabPanel("Trend", plotOutput("trend"),
|   downloadButton("downloadData.trend", "Download graphic"))
```

As you can see, I have added the button underneath the graph, so users know what they are downloading.

Downloadable reports with knitr

This same function can very easily allow your users to produce custom reports in HTML, pdf, or MS Word ready to be downloaded to their machines, using the `knitr` package (<http://yihui.name/knitr/>). `knitr` is a user-contributed package that allows reports to be generated dynamically from a mixture of a static report formats interleaved with the output from of R.

So, for example, titles and text can be fixed, but each time the report is run, a different output will be produced within the document depending on the state of the data when the output is generated. `knitr` can be used with the R Markdown format. Here is the simple R Markdown document within the Google Analytics application:

```
# Summary report
## Text summary
This report summarises data between `r strftime(input$dateRange[1],
  format = "%d %B %Y")` and `r strftime(input$dateRange[2],
  format = "%d %B %Y")`.

## Trend graph
```{r fig.width=7, fig.height=6, echo=FALSE}
trendGraph()
````
```

As can be seen, the document is a mix of static headings and text, inline R output (given as ``r "print("somthing")``), and graphical output. The `trendGraph()` function, of course, is the same `trendGraph()` function that we saw in the download graphics code.

The code to download the report is as follows (with the R Markdown document in the same folder as `server.R` and named `Report.Rmd`):

```
output$downloadDoc <-
  downloadHandler(filename = "Report.html",
    content = function(file){
      knit2html("Report.Rmd", envir = environment())

      # copy document to 'file'
      file.copy("Report.html", file,
        overwrite = TRUE)
    }
  )
```

Adding a button to download the graph is the same as for the downloading graph function; the following should be placed in `ui.R` within the `sidebarPanel()` function:

```
| downloadButton("downloadDoc", "Download report")
```

Downloading and uploading data

Downloading data is done in a similar fashion, which looks like the following `downloadHandler()` call:

```
| output$downloadData <- downloadHandler(  
|   filename = function(){  
|     "myData.csv"  
|   }  
|   content = function(file){  
|     write.csv(passData(), file)  
|   }  
)
```

Uploading data is achieved using the `fileInput()` function. In the following example, we will assume that the user wishes to upload a comma-separated spreadsheet (`.csv`) file. The button is added to `ui.R` in the following manner:

```
| fileInput("uploadFile", "Upload your own CSV file")
```

This button allows a user to select their own `.csv` file, and it also makes a variety of objects based on the ID (in this case, `input$uploadFile$`) available from `server.R`. The most useful is `input$uploadFile$datapath`, which is a path to the file itself and can be turned into a dataframe using `read.csv()`:

```
| userData <- read.csv(input$uploadFile$datapath)
```

There are other bits of information about the file available. Take a look at `?fileInput` for more details.

Bookmarking

Internet users are familiar with the term bookmarking. If we are surfing a site and think it's something important that we'd like to visit again, we usually bookmark it. How about the bookmarking a Shiny app? As the Shiny app has data representation and input/output interactions, saving this state using bookmarking and then reproducing it will require some coding expertise.

In this section, we are going to focus on how to bookmark the state of a Shiny app. We will see coding modifications in the basic structure of the Shiny app needed to incorporate the bookmarking feature. We will explore bookmarking a state, and some exceptional cases and how to deal with them.

Bookmarking state

The state of an application can be defined as the state of input and output at a given time. In Shiny apps, the state of the input and output can be the input in the input box and the output in the output box or a graph at a particular time. Here, our aim is to save a state at a given time and restore it. As of now we have learned about the structure of Shiny apps. Shiny apps can be a single file (UI and Server file together) or multiple files (UI, Server or global files separately). And there can be apps where the flow of input and output can be straightforward or not straightforward. A straightforward control flow of an app can be considered as the app that has a deterministic output for an input and vice-versa. In this section, we will learn to program the bookmarking features for the state of apps with a straightforward control flow.

The methods of bookmarking can be classified into two categories:

- Bookmarking by Encoding the state into a URL
- Bookmarking by saving the state to the server

In both the methods, the basic concept is to preserve the state of the application at a particular time. The difference lies in how to save the state. Let's discuss both the methods in detail.

Encoding the state into a URL

In this method of bookmarking the state, the state of the app is embedded into a URL. As we have discussed, the state of the application means the input and output values at a particular time. So, the input and output values are embedded in the app for a time instance. The input and output values can be seen in the URL itself. As we are familiar with the coding of Shiny apps, the differences with bookmarking are that the UI has to be returned as a value of a function with a single argument and `enableBookmarking()` has to be called. When we discussed the different styles of coding a Shiny app in single file or multiple files, we learned we need to understand how to implement the requirements-implementing UI with a single argument function and using `enableBookmarking()`.

Single-file application

Single-file apps have UI and server code in the same file. For such a coding style, the app skeleton can be given as follows:

```
| ui<-function(request){# UI code }
| server<- function(input,output,session){# Server Code}
| shinyApp(ui,server,enableBookmarking="url")
```

We can see that the UI code is returned as a function rather than `fluidpage` or any other UI element. And in the last line of code, the `enableBookmarking` argument is set to "url". The rest of the code is similar to Shiny apps. Let's develop an app with the preceding skeleton that finds the square of a number. Starting with UI, there is a `textInput()` box for getting the input, `verbatimTextOutput()` for showing output, and `bookmarkButton()` for bookmarking:

```
| ui<-function(request){
|   fluidPage(
|    textInput("inptxt","Enter Number"),
|     verbatimTextOutput("outsquare"),
|     bookmarkButton())
| }
```

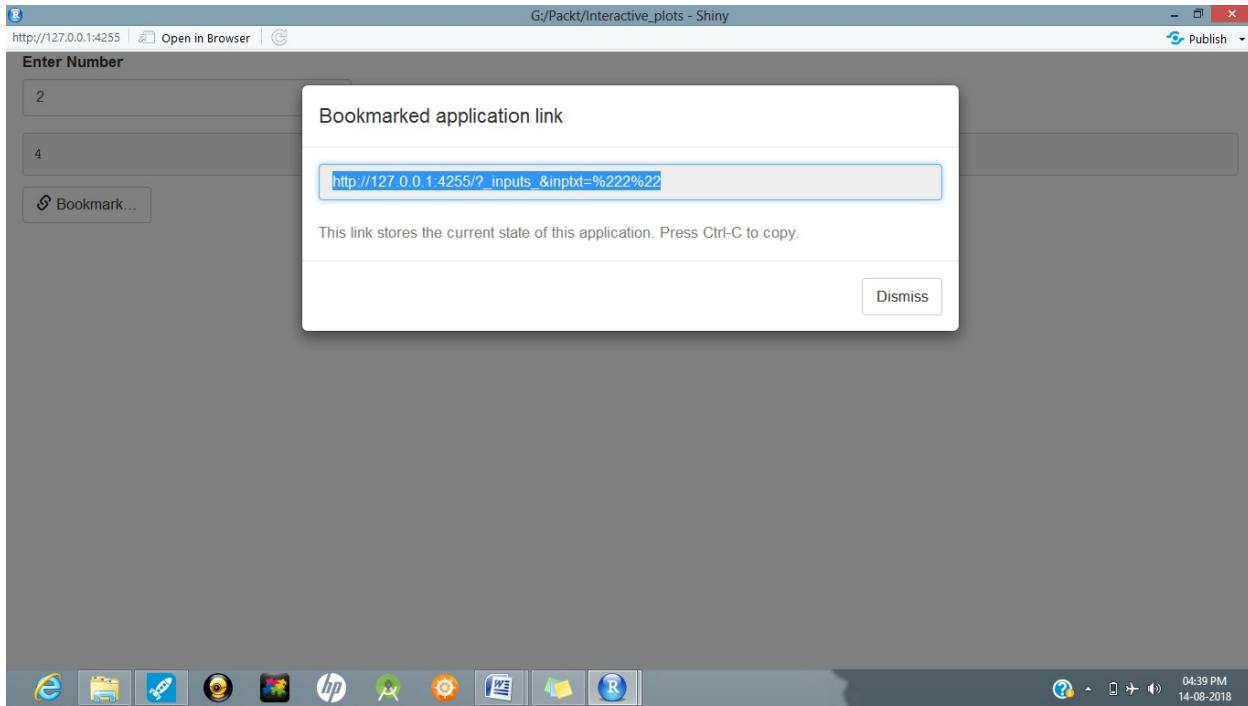
In the server code, we have to calculate the square of the inputted number and send it for display. We can see in the following code that `renderText()` is first converting the inputted text into a number and then calculating the square:

```
| server<- function(input,output,session){
|   output$outsquare<- renderText({
|     (as.numeric(input$inptxt)*as.numeric(input$inptxt))
|   }) }
```

In the last, we have to call `shinyApp()` in which we are passing the `enableBookmarking` argument as "url":

```
| shinyApp(ui,server,enableBookmarking="url")
```

Let's have a look at output of the code:



We can see that in the app, the input is 2 and the output value is 4. The bookmark window shows the linked URL with some values that are encoding the input/output, `http://127.0.0.1:4255/?_inputs_&inptxt=%222%22`.

Single file Shiny applications can also be returned by a function. Let's recreate the square application using `function`:

```
App<-function(){
  ui<-function(request){
    fluidPage(
     textInput("inptxt","Enter Number"),
      verbatimTextOutput("outsquare"),
      bookmarkButton()
    )
  }
  server<- function(input,output,session){
    output$outsquare<- renderText({
      (as.numeric(input$inptxt)*as.numeric(input$inptxt))
    })
  }
  shinyApp(ui,server,enableBookmarking="url")
}
```

To run the app, copy the code in to an R Script file, select all code, and run it. After that, we can execute the code using `App()` in console or R Script. `enableBookmark()` can also be set at the beginning of the app code.

Now, we are ready to develop our application for multiple files.

Multiple-file application

In Shiny applications where UI and server codes are kept in separate files, `enableBookmarking()` has to be kept in the `global.r` file. The rest of the things are the same. For redeveloping the squared-number app in a multiple-file pattern, we have to copy the UI code and paste it into the `ui.r` file and server code in the `server.r` file with `enableBookmarking(store="url")` in the `global.r` file. We will get the same output as we got in the single-file implementation.

Bookmarking by saving the state to the server

In this method, the only modification in terms of coding is to change the `enableBookmarking` argument to the server. Let's recreate the square-calculation application using saving state to the server:

```
ui<-function(request){
  fluidPage(
   textInput("inptxt","Enter Number"),
    verbatimTextOutput("outsquare"),
    bookmarkButton()
  )
}
server<- function(input,output,session){
  output$outsquare<- renderText({
    (as.numeric(input$inptxt)*as.numeric(input$inptxt))
  })
}
shinyApp(ui,server,enableBookmarking="server") # saving to the server
```

How does it make a difference in processing? Saving state to the server method saves the state of the application on the server instead of embedding the states into a URL. Shiny provides two versions of servers. One is the open source Shiny server, and the other is Shiny server pro. In Shiny server, the state of the application is saved in the `/var/lib/shiny-server/bookmarks` subdirectory. RStudio Connect also stores the bookmark state in `/var/lib/rstudio_connect/bookmarks`. We usually experiment with our applications on desktop versions of Rstudio. In such cases, the state is saved in the `shiny_bookmarks/` subdirectory.

The advantage of using saving state in the server is that files can also be bookmarked, which is not possible with bookmarking with encoding. Large files are not supported by bookmarking with a URL. To save state on the server, the hosting environment must be supported.

As of now we have discussed bookmarking state for the apps having straightforward flow. But if apps don't have straightforward flows, we need to take some extra measures to deal with such situations. In a similar way, if apps have random-number generation and use that to generate output, then these methodologies won't work.

For such applications with complex state, use of callback functions is needed. Shiny has provided `onBookmark()`, and `onRestore()` to achieve callback functions. These callback functions take one argument, named `state`. We can use `state$value` to save and retrieve values to be bookmarked.

Let's develop an application that generates a random number and adds it to the inputted number. Here is the UI code for getting the input number using `textInput()`. It has one `actionButton()` for triggering the event of the addition, and `bookmarkButton()` for bookmarking:

```
ui <- function(request) {
  fluidPage(
    sidebarPanel(
      textInput("txt", "Number"),
      actionButton("add", "Add"),
      bookmarkButton()
    ),
    mainPanel(
      h4("Sum of Random Number and Inputed Number:", textOutput("result"))
    )
  )
}
```

In the server code, we can see that the result variable is initiated to zero and a random-number generator, `runif()`, has been used. `onBookmark()` is used to save the result value in the `state$values` object. And `onRestore()` is using the same `state` object to restore the bookmarked value:

```
server <- function(input, output, session) {
  vals <- reactiveValues(result = 0)

  # Save values in state$values for bookmark
  onBookmark(function(state) {
    state$values$currentresult <- vals$result
  })

  # Read values from state$values when we restore
  onRestore(function(state) {
    vals$result <- state$values$currentresult
  })

  # Exclude the add button from bookmarking
  setBookmarkExclude("add")

  observeEvent(input$add, {
    vals$result <- vals$result + floor(runif(1))+as.numeric(input$txt)
  })
  output$result <- renderText({
    vals$result
  })
}

shinyApp(ui, server, enableBookmarking = "url")
```

So, we have learned how to bookmark the Shiny app state in a simple, straightforward flow as well as in complex states. We also learned in which conditions encoding with a URL is to be used and when a save on the server can be advantageous.

Interactive plots

To explore complex patterns in data, simply putting data into a static graph isn't particularly useful. In the big data era, interactive plots demand a lot of time. R has come up with very elegant interactive options:

- click
- double-click
- hover
- brush
- zoom in/out
- selection of area

`plotOutput()`, like function, provides options to enable these properties to make plots more interactive. Let's explore `plotOutput()` and use it in our app. Here is the syntax for `plotOutput()`:

```
| plotOutput(outputID, width, height, click=NULL, dblclick=NULL, hover=NULL, hoverDelay=NULL, hoverD
```

Let's develop an application that explores the iris dataset and uses some interactive features of plotting:

```
library(shiny)
shinyApp(
  ui = basicPage( fluidRow(
    column(width = 4,
      plotOutput("plot", height=300,
        click = "plot_click",
        hover = hoverOpts(id = "plot_hover", delayType = "throttle"),
        brush = brushOpts(id = "plot_brush"))
    ),
    h4("Clicked points"),
    tableOutput("plot_clickedpoints")
  ),
  column(width = 4, verbatimTextOutput("plot_hoverinfo"))
),
server = function(input, output, session) {
  output$plot <- renderPlot({
    plot(iris$Sepal.Length, iris$Sepal.Width)
  })
}
```

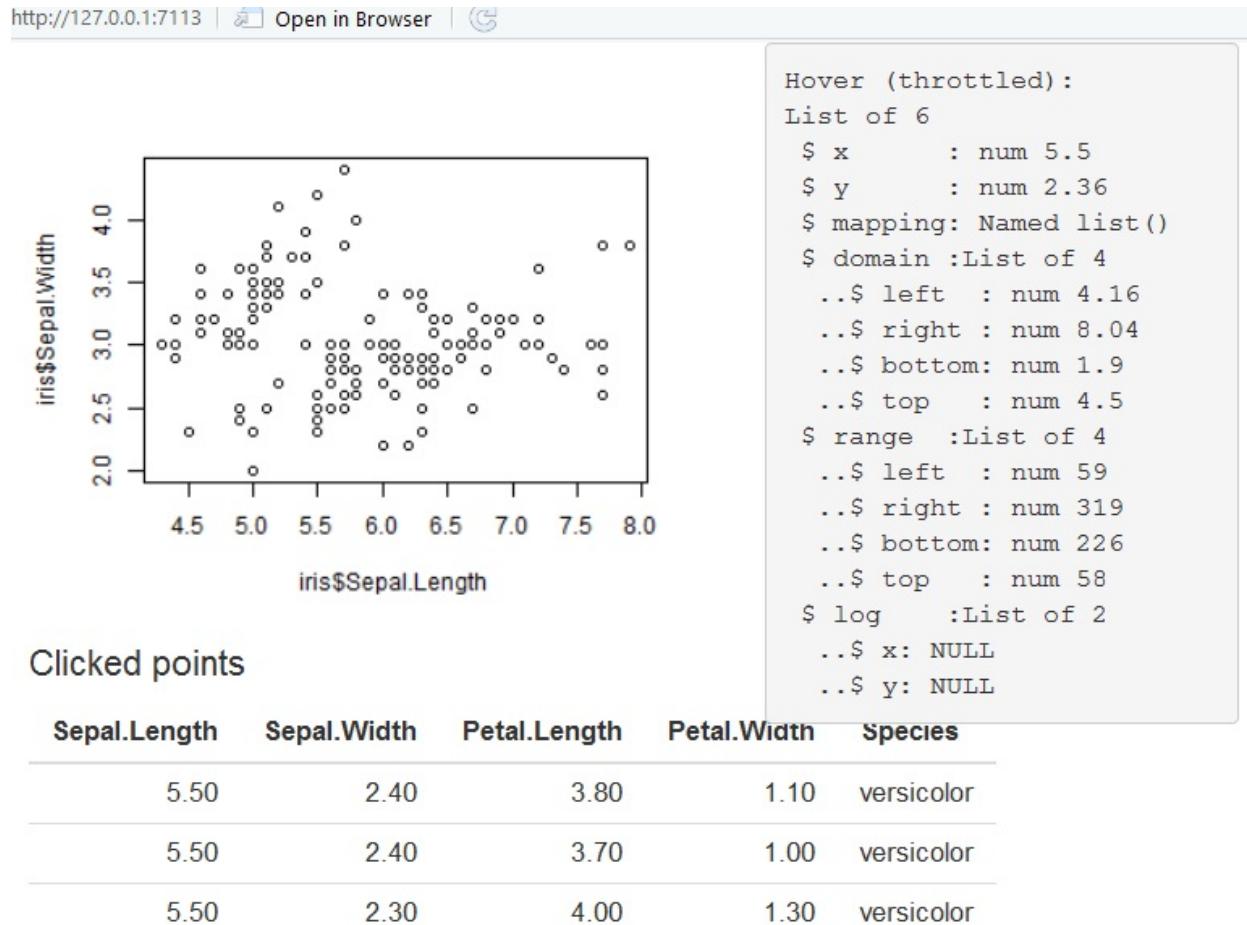
```

    output$plot_clickedpoints <- renderTable({
      # For base graphics, we need to specify columns, though for ggplot2,
      # it's usually not necessary.
      res <- nearPoints(iris, input$plot_click, "Sepal.Length", "Sepal.Width")
      if (nrow(res) == 0)
        return()
      res
    })
    output$plot_hoverinfo <- renderPrint({
      cat("Hover (throttled):\n")
      str(input$plot_hover)
    })
  }
)

```

In this code, we can see that `plotOutput()` has the click and hover properties. The `click` property specifies that it will detect a click event on the plot. `hover` will detect the pointer's position.

In the following output, the position of pointer is captured and details are displayed in the hover window and data for the point is in the table:

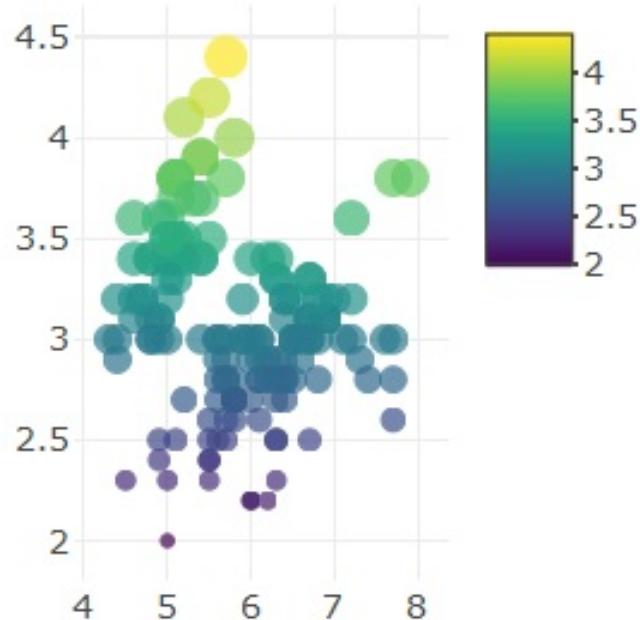


`ggplot` and `plotly`, like R libraries, are full of very good interactive plotting functions. These functions use `d3.js` to produce interactive plots and graphs. As we have already discussed `ggplot2`, we will not elaborate on it here.

`plotly` is also very easy to use. If `plotly` is not installed on your system, you can install it using `install.package("plotly")`. Let's look at an example with `plotly`:

```
library (plotly)  
plot_ly(iris, x = iris$Sepal.Length, y = iris$Sepal.Width, text = paste("iris$Sepal.Width"),  
       mode = "markers", color = iris$Sepal.Width, size = iris$Sepal.Width)
```

In the first line of code, the `plotly` library is invoked which facilitates the use of `plot_ly()`. In `plot_ly()`, the first argument is the name of the dataset, the second is the `x` axis, and the third is the `y` axis. The output can be seen in the following screenshot; by default, it supports hover and click events:



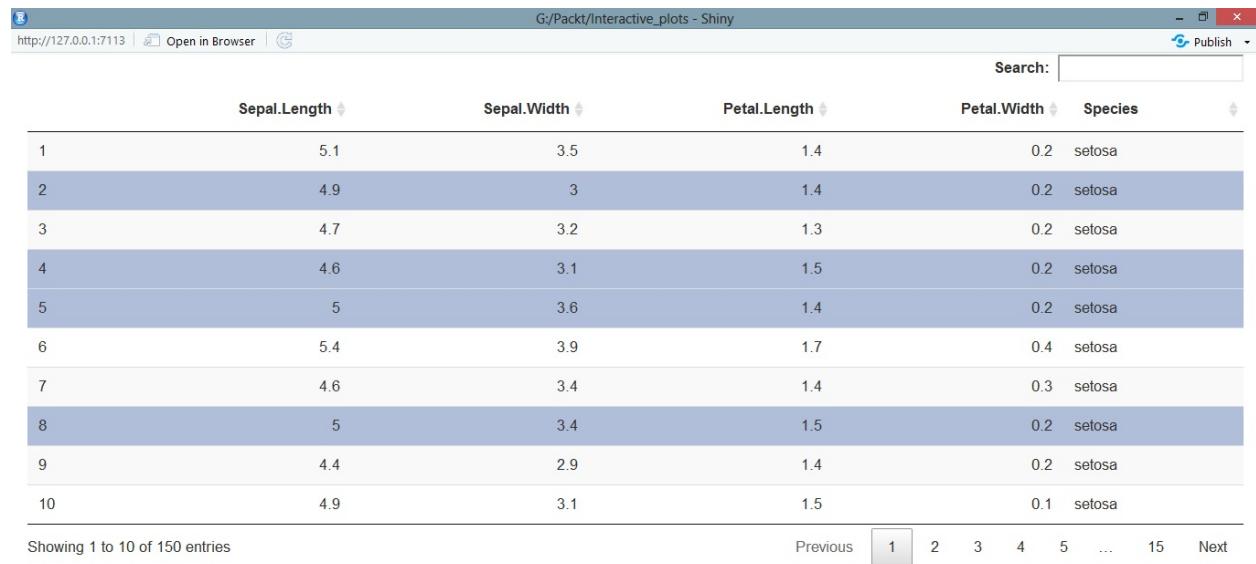
Interactive tables

In the preceding example application, we saw how to make our plot interactive. Now it's time to make tables interactive. Interactive plots mean we can have freedom in selecting rows/columns/cells and shorting columns.

`DT` is a package with `DT` library that helps us to make tables more interactive. Let's see a small client-side example of Data Tables with Shiny:

```
library(shiny)
library(DT)
shinyApp(
  ui = fluidPage(DTOutput('tbl')),
  server = function(input, output) {
    output$tbl = renderDT(
      iris, options = list(lengthChange = FALSE)
  )
})
```

In the preceding code, `DToutput()` is for outputting the table on UI, and on the server side we have used `renderDT()` to send the data into the UI from the `iris` dataset. The output can be seen here:



The screenshot shows a web browser window titled "G/Packt/Interactive_plots - Shiny". The URL is "http://127.0.0.1:7113". The page displays a data table with the following columns: Sepal.Length, Sepal.Width, Petal.Length, Petal.Width, and Species. The table has 150 entries. The first 10 entries are shown in the table below:

| | Sepal.Length | Sepal.Width | Petal.Length | Petal.Width | Species |
|----|--------------|-------------|--------------|-------------|---------|
| 1 | 5.1 | 3.5 | 1.4 | 0.2 | setosa |
| 2 | 4.9 | 3 | 1.4 | 0.2 | setosa |
| 3 | 4.7 | 3.2 | 1.3 | 0.2 | setosa |
| 4 | 4.6 | 3.1 | 1.5 | 0.2 | setosa |
| 5 | 5 | 3.6 | 1.4 | 0.2 | setosa |
| 6 | 5.4 | 3.9 | 1.7 | 0.4 | setosa |
| 7 | 4.6 | 3.4 | 1.4 | 0.3 | setosa |
| 8 | 5 | 3.4 | 1.5 | 0.2 | setosa |
| 9 | 4.4 | 2.9 | 1.4 | 0.2 | setosa |
| 10 | 4.9 | 3.1 | 1.5 | 0.1 | setosa |

Showing 1 to 10 of 150 entries

Previous 1 2 3 4 5 ... 15 Next



We can see in the preceding screenshot that we can select rows using a mouse pointer. In this way, interactive functionality can be added to tables.

`DT` supports both server-side and client-side processing; the default is server-side, but client-side can be set by calling `DT::renderDT()` with a `server = FALSE` argument. If the dataset is relatively small, use `server = FALSE`, otherwise it will be too slow to render the table in the web browser, and the table will not be very responsive, for a large data set. Example code:

```
| DT::renderDataTable(iris, server = FALSE)
| For more details https://rstudio.github.io/DT/ link can be followed.
```

Row selection

Row selection is the default in `datatable(...,)`. We can toggle the selection by clicking on the row. It can also be disabled by setting the `datatable(..., selection = 'none')` property. The 'selection' mode can also be set to 'single' or 'multiple'. By default, the multiple selection mode is set.

Column selection

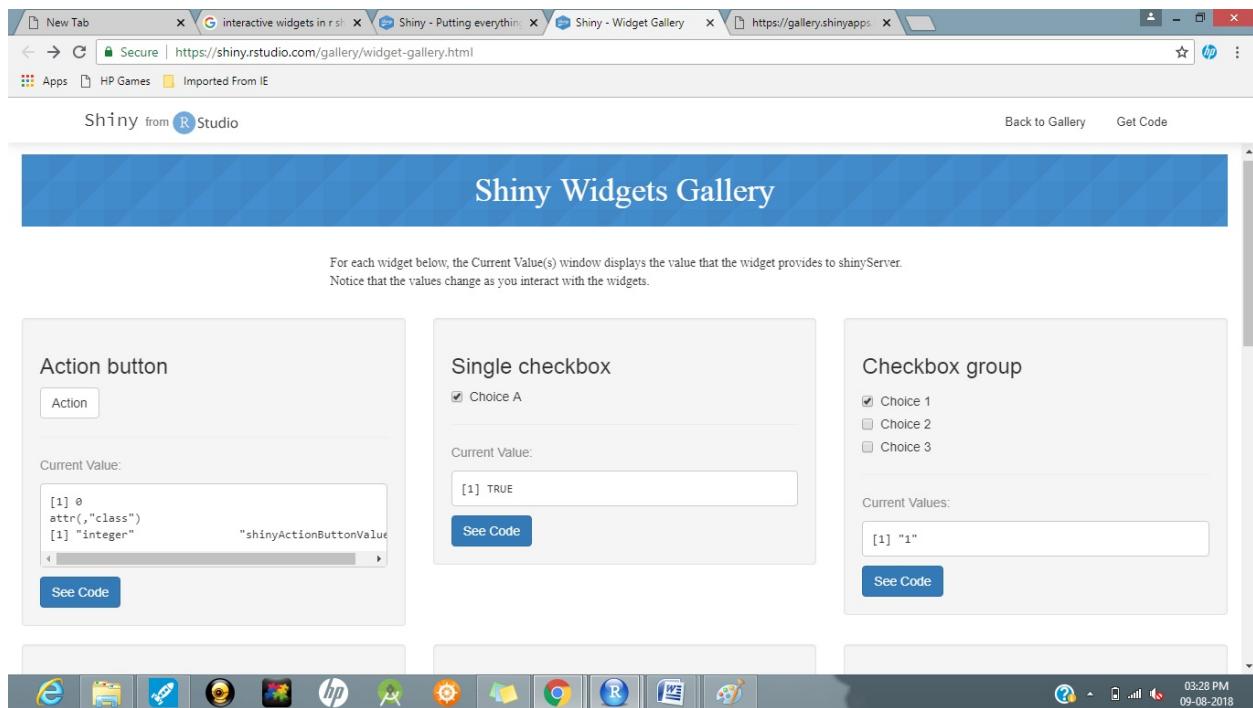
To change the default mode row selection to column, we need to set the target as column in datatable as `datatable(..., selection = list(target = 'column'))`. To select row and column simultaneously, the target can be set as "`row+ column`".

Cell Selection

To enable cell selection, the value of `target` must be set to `cell`.

Linking interactive widgets

Shiny has a number of widgets that can be linked in our applications. For referring to the widget gallery go to <https://shiny.rstudio.com/gallery/widget-gallery.html>. Here, you can find widgets with sample output and a see code button. Have a look at a snapshot of the site:



For better understanding, we will build an application with a Shiny dashboard and integrate widgets into our app. Let's develop our app step by step:

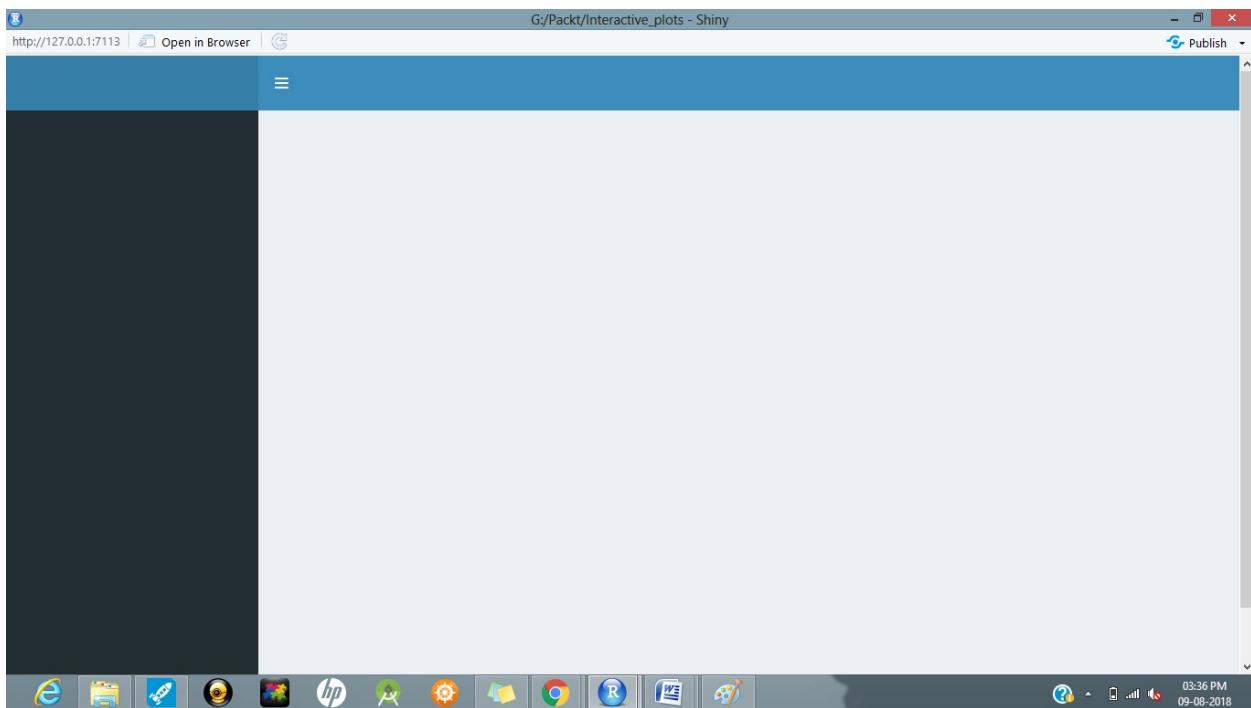
1. Develop a basic dashboard.
2. Copy the code into an R Script file.
3. If the `shinydashboard` package is not installed, install it.
4. Execute the code by clicking on the run app button:

```
## app.R ##
library(shiny)
library(shinydashboard)

ui <- dashboardPage(
  dashboardHeader(),
  dashboardSidebar(),
  dashboardBody()
```

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

Here is the output:



Now, let's integrate the widget into our basic dashboard:

1. Go to <https://shiny.rstudio.com/gallery/widget-gallery.html>.
2. Let's integrate the File input widget into our app. So click on File input to see the code.
3. Copy the code from the UI file and paste it into the `dashboardbody()` dashboard UI files:

```
dashboardBody(  
  fileInput("file", label = h3("File input")),  
  hr(),  
  fluidRow(column(4,  
    verbatimTextOutput("value")))  
)
```

4. Copy the code for the server file of the File input widget and paste in the server code of our dashboard:

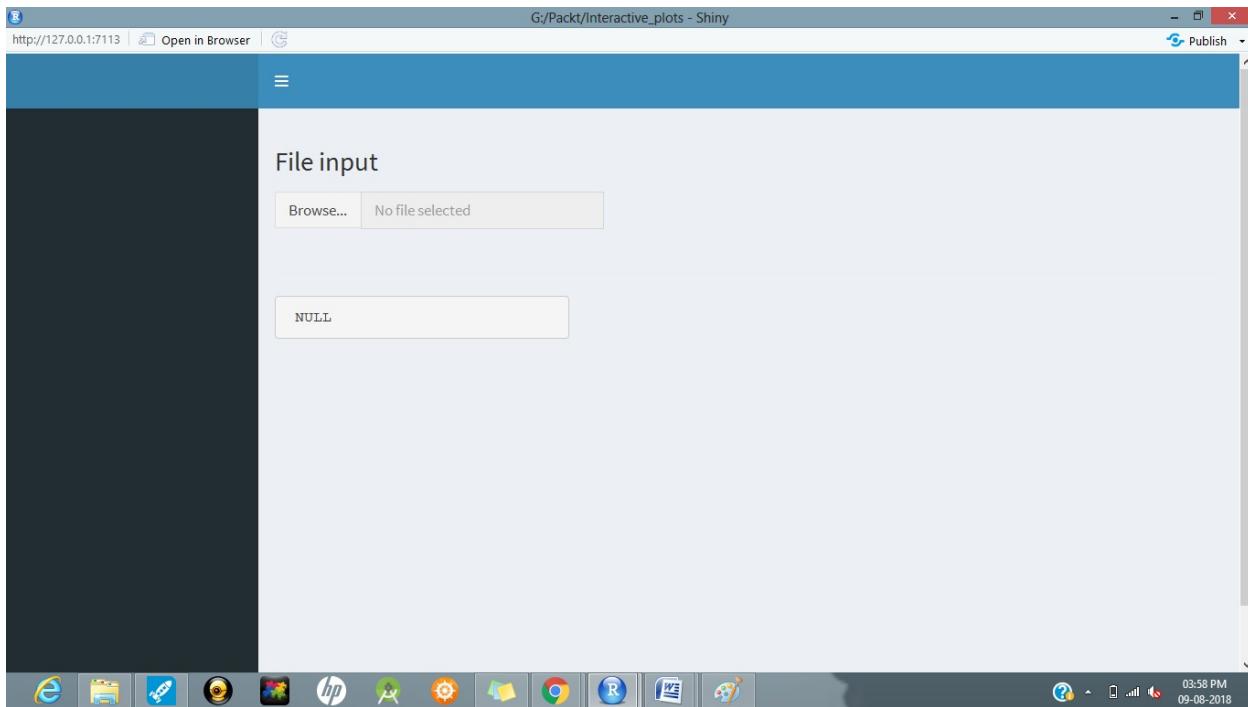
```
server <- function(input, output) {  
  output$value <- renderPrint({  
    str(input$file)
```

```
|   })  
| }  
| }
```

5. Our file-upload manager app is ready. The complete code can be seen here:

```
## app.R ##  
library(shiny)  
library(shinydashboard)  
ui <- dashboardPage(  
  dashboardHeader(),  
  dashboardSidebar(),  
  dashboardBody(fileInput("file", label = h3("File input")),  
                hr(),  
                fluidRow(column(4, verbatimTextOutput("value"))))  
)  
server <- function(input, output) {  
  output$value <- renderPrint({  
    str(input$file)  
  })  
}  
shinyApp(ui, server)
```

Here is the output:



From the preceding screenshot, we can see that the File input widget has been integrated into our application, and after clicking on the Browse... button, any file can be uploaded.

Shiny gadgets

So far, we have seen how to present data with Shiny apps for the end user. Now it's time to develop gadgets for R coders. The main difference between Shiny apps and Shiny gadgets is that apps are deployed on servers, such as `shinyapps.io` or a deployable server, but gadgets are to be called from the code from R Script. At the same time, Shiny gadgets can be registered with RStudio as add-ins.

Shiny gadgets have been developed to make repeatable tasks easily doable, such as importing data in the right formats, cleaning data, manipulating, or visualization. Now let's see how to write Shiny gadgets:

```
library(shiny)
library(miniUI)

newGadget <- function(inputVal1, inputVal2) {

  ui <- miniPage(
    gadgetTitleBar("New Gadget"), # title of Gadget
    miniContentPanel(
      # layout, inputs, outputs
    )
  )

  server <- function(input, output, session) {
    # Define reactive expressions, outputs, etc.

    # When the Done button is clicked, return a value
    observeEvent(input$done, {
      returnValue <- ...
      stopApp(returnValue)
    })
  }

  runGadget(ui, server)
}
```

From the gadgets skeleton, we can see that it is similar to Shiny apps. The packaging of the UI and server logic is done differently in gadgets. Shiny apps are generally kept in the app's directory with the server and UI files, but Shiny gadgets are defined inside a regular function.

As we can see from the skeleton, we still have the UI and server construct, but they're all wrapped in a regular function. And hence the function's `inputValue1` and `inputValue2` arguments can be accessed locally, and return values from the server

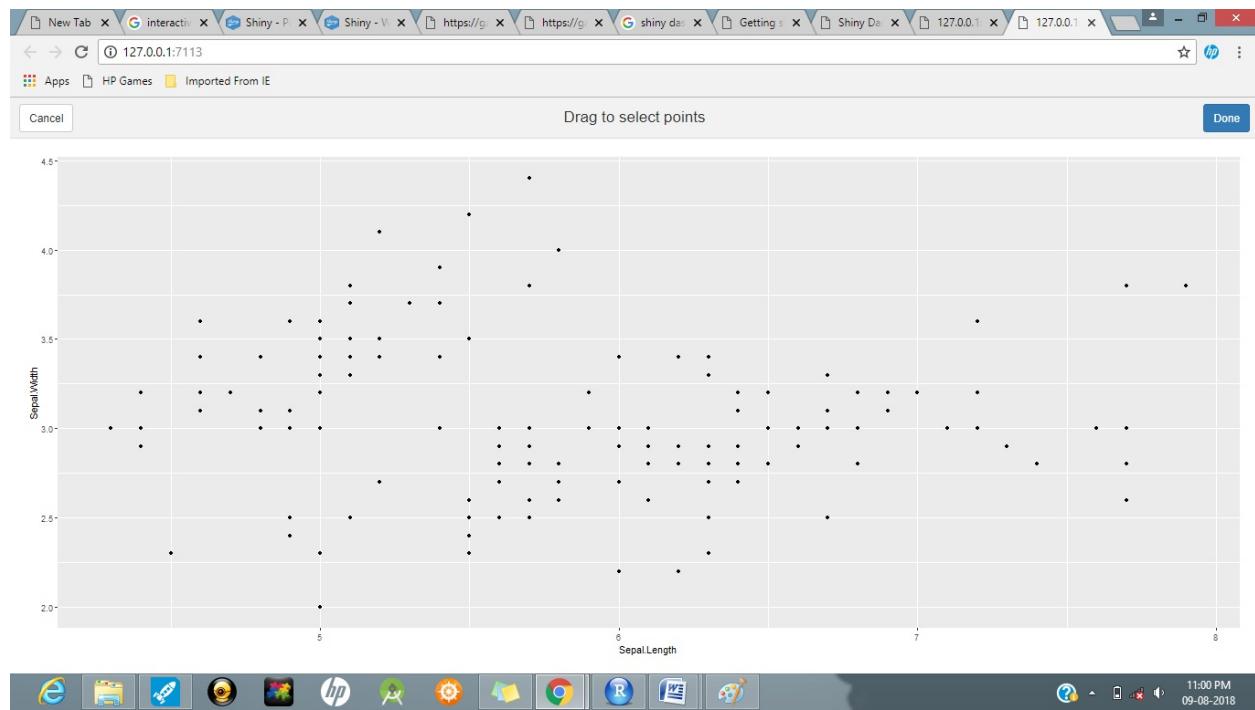
construct can be the return value of the gadget.

Again in skeleton, UI has `miniPage` instead of `fluidpage`, and `miniContentPanel` in place of `sidelayout`. This is because the gadget's output will open in a separate RStudio Viewer pane instead of a web page. The `MiniUI` construct is used to make optimal use of available space.

Let's develop a Shiny gadget to plot the data passed as an argument, detect the click event, and show the data:

```
library(shiny)
library(miniUI)
library(ggplot2)
click_gadget <- function(data, xvar, yvar) {
  ui <- miniPage(
    gadgetTitleBar("Drag to select points"),
    miniContentPanel(
      plotOutput("plot", height = "100%", click = "plot_click")
    )
  )
  server <- function(input, output, session) {
    # Render the plot
    output$plot <- renderPlot({
      # Plot the data with x/y vars indicated by the caller.
      ggplot(data, aes_string(xvar, yvar)) + geom_point()
    })
    # Handle the Done button being pressed.
    observeEvent(input$done, {
      # Return the brushed points. See ?shiny::brushedPoints.
      stopApp(clickOpts(data, input$plot_click))
    })
  }
  runGadget(ui, server)
}
```

To execute this code, first select all the preceding code in R Script and execute it, then run `click_gadget(iris, "Sepal.Length", "Sepal.Width")`. Here, the `iris` dataset has been used, but it can be changed to any dataset:



At the same time, if the plot is clicked and the Done button is pressed, we can see the data of the inputted dataset.

Adding a password

So far, we have learned how to develop a Shiny application. Since the application is exposing so much data to the outside world, it needs to be protected by unauthorized means. For that, we can provide password authentication. Shiny provides a control for achieving this task, called `passwordInput` (<https://shiny.rstudio.com/reference/shiny/latest/passwordInput.html>):

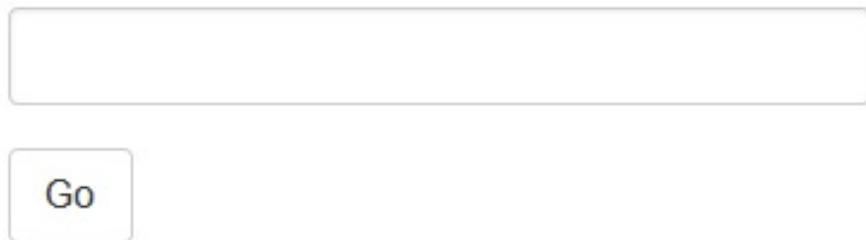
```
| passwordInput(inputId, label, value = "", width = NULL,placeholder = NULL)
```

Let's see an example with `passwordInput`:

```
ui <- fluidPage(  
  passwordInput("password", "Password:"),  
  actionButton("go", "Go"),  
  verbatimTextOutput("value")  
)  
server <- function(input, output) {  
  output$value <- renderText({  
    req(input$go)  
    isolate(input$password)  
  })  
}
```

Here is the output:

Password:



In the preceding application, the UI section has `passwordInput()`, which converts the entered text into dots and the inputted value is stored in a `password` variable that can further be used in reactive processes. The code can be copied and pasted into R Script and executed as a simple Shiny application.

Summary

In this chapter, we learned about R Shiny's advanced features. Animation, graphics, report development, and using `knitr` are wonderful professional-level presentation features; using these, anybody can share their results and tell the story about the data in hand. We also discussed and demonstrated the feature of bookmarking the Shiny app's state with different levels of complexity.

Interactive plots and tables are new ways to play with data and finding new ways of dynamic presentation. Widgets are like cookbooks and can be used to speed up the app-development process. We can now also code for R users and developers by developing gadgets. Adding a password to our app can give us extra control on restricting the data accessed by end users. In this chapter, we studied most of the latest advanced features of presentation, code sharing, and authentication.

Code Patterns in Shiny Applications

As your Shiny applications become larger and more complex, it is important to write clear, readable, and maintainable code. In some cases, it is also necessary to modularize your code. This chapter covers various aspects of writing and debugging Shiny applications, functions, and modules. We will also look at features that relate to the control of reactivity within your application and how to speed up Shiny applications.

In this chapter, we will cover the following topics:

- Reactivity in RShiny
- Controlling specific input with the `isolate()` function
- Running reactive functions over time
- Event handling using `observeEvent`
- Functions and Modules
- Shinytest
- Debugging
- Handling errors (including `validate()` and `req()`)
- Profiling R code
- Debounce and throttle

Reactivity in RShiny

RShiny uses a reactive programming coding pattern, which makes applications responsive. In reactive programming, there are basically three elements:

- Reactive Source
- Reactive Endpoint
- Reactive Conductor

The combination of these elements is what makes a Shiny app more responsive. Let's have a look in more detail at how they work:

- **Reactive Source:** The reactive source is the input objecting the Shiny app. It takes the input from the user and keeps it in the `input` object. The `input` object is associated with any UI element, with which the user interacts and provides input. It can be an input box, an action button, an interactive table or plot, or any other UI component. For example, if we take input from a text box named `txtbox` in the UI code, we can say that `input$txtbox` is the reactive source.
- **Reactive Endpoint:** The reactive endpoint is the output object in a Shiny app. It accepts a value and puts that in the output components of the UI. The output components can be any UI element that is used to display the data in any format. The output object is `output`. For example, if we want to put the data into a plot named `hist_plot` in the UI file, then `output$hist_plot` will be the reactive endpoint.

A simple application can contain only these two elements: the source and the endpoint. Let's take a look at an example of an application that usually appears whenever we start a new Shiny application:

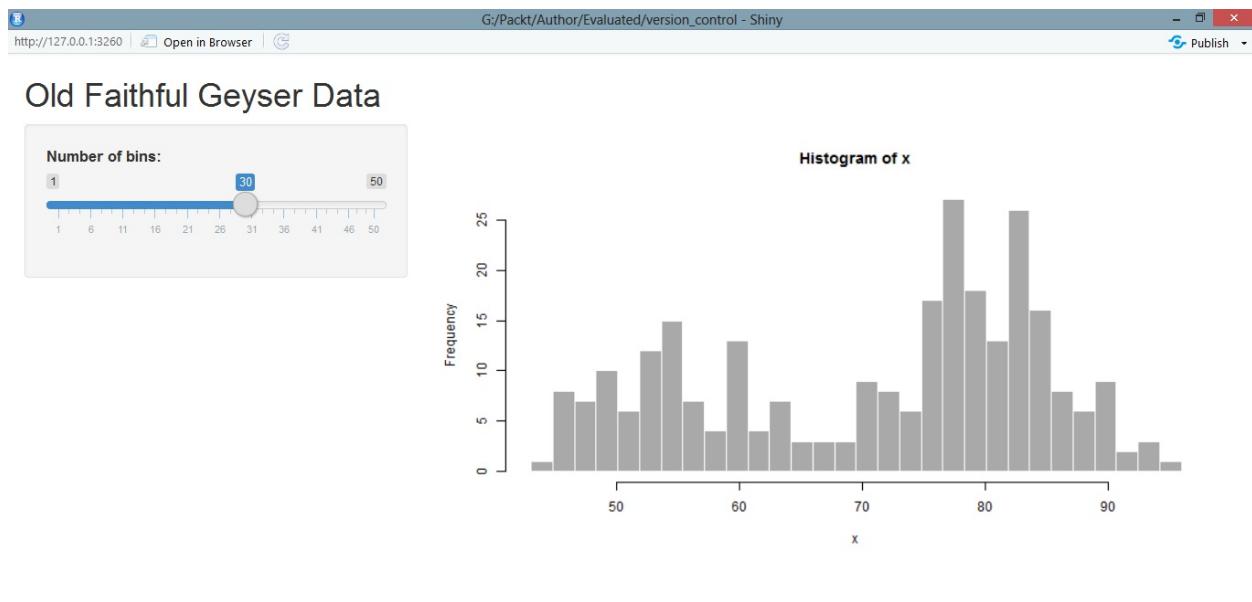
```
library(shiny)
# Define UI for application that draws a histogram
ui<- fluidPage(
  # Application title
  titlePanel("Old Faithful Geyser Data"),
  # Sidebar with a slider input for number of bins
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                 "Number of bins:",
```

```

min = 1,
max = 50,
value = 30)
),
# Show a plot of the generated distribution
mainPanel(
plotOutput("distPlot")
)
)
)
# Define server logic required to draw a histogram
server<- function(input, output) {
output$distPlot<- renderPlot({
# generate bins based on input$bins from ui.R
x      <- faithful[, 2]
bins<- seq(min(x), max(x), length.out = input$bins + 1)
# draw the histogram with the specified number of bins
hist(x, breaks = bins, col = 'darkgray', border = 'white')
})
}
# Run the application
shinyApp(ui = ui, server = server)

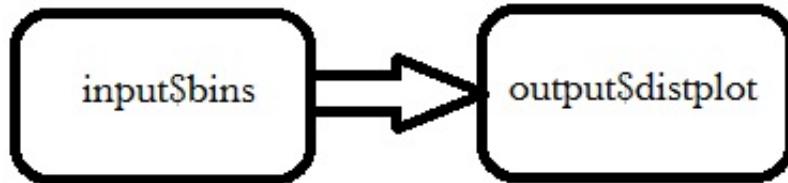
```

You get the following output:

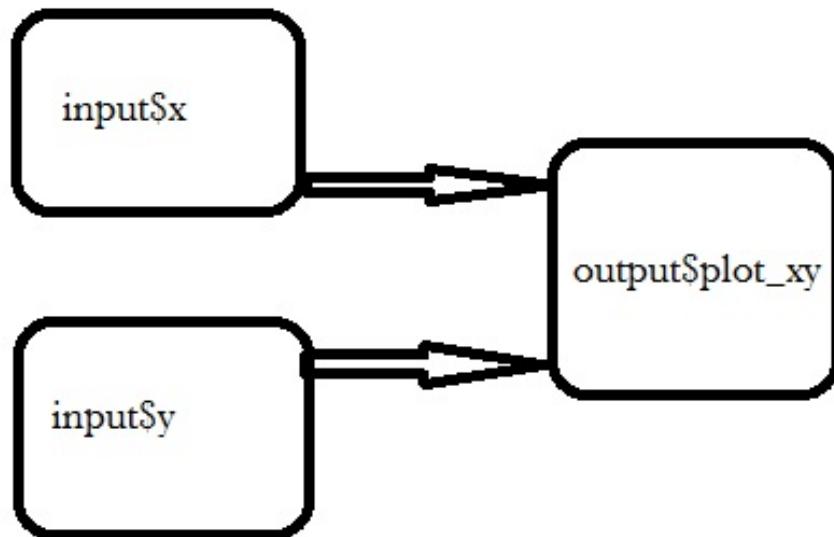


In the UI section of the preceding code, `sliderInput` is the input component named `bins`. In the server code section, `input$bins` is the reactive source. Similarly, `plotOutput("distPlot")` is the output component on the UI. As the value of `input$bins` changes, this plot also changes. This is how reactivity works. In the `server`

section, `output$distPlot` is the reactive endpoint. This is represented here:



However, with more complex apps, the reactive source and endpoints can form different combinations. Imagine that a graph is to be built using two input values. In this case, the flow diagram would be as follows:



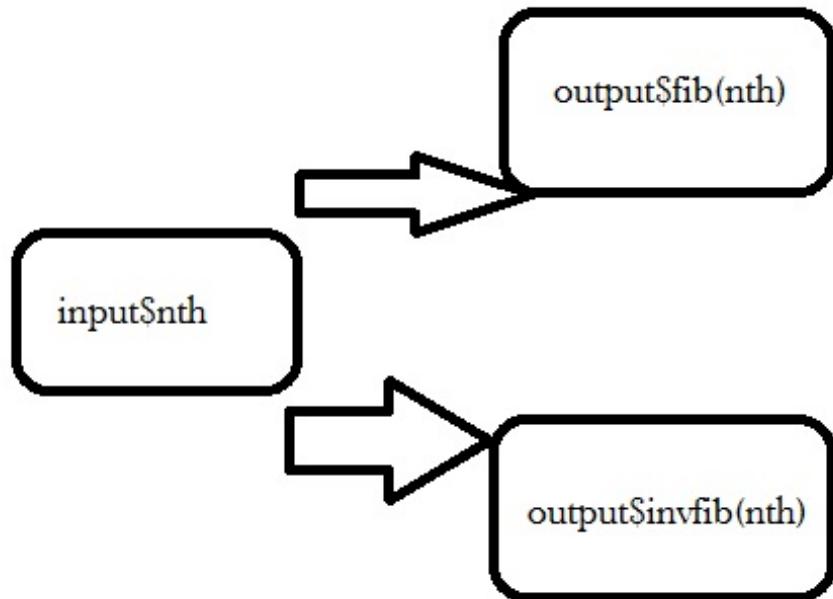
- **Reactive Conductor:** So far, we have seen that the source is directly connected to the end point. There may be situations where there is a middle element between these two. This middle element is called a reactive conductor.

Reactive conductors are usually used to encapsulate slow operations or catch data. From Shiny's official website, <https://shiny.rstudio.com/articles/reactivity-overview.html> to calculate a number and inverse of it in the Fibonacci sequence for the

n^{th} element of a single application, which can be very time-consuming. This situation is shown in the code:

```
# Calculate nth number in Fibonacci sequence
fib<- function(n) ifelse(n<3, 1, fib(n-1)+fib(n-2))
server<- function(input, output) {
  output$nthValue<- renderText({ fib(as.numeric(input$n)) })
  output$nthValueInv<- renderText({ 1 / fib(as.numeric(input$n)) })
}
```

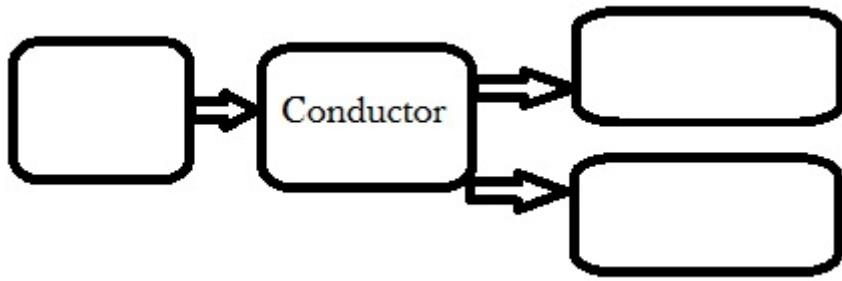
The flow diagram would be as follows:



In such situations, a middle layer to catch the results can be added to reduce the time taken by this process. That is called a reactive conductor. The preceding code can be modified using a reactive conductor, as follows:

```
fib<- function(n) ifelse(n<3, 1, fib(n-1)+fib(n-2))
server<- function(input, output) {
  currentFib<- reactive({ fib(as.numeric(input$n)) })
  output$nthValue<- renderText({ currentFib() })
  output$nthValueInv<- renderText({ 1 / currentFib() })
}
```

Now the flow diagram will look as follows:



Up until now, we have discussed three elements: the reactive source, the reactive conductor, and the reactive endpoints. These are the general terms used to describe reactive programming elements. In RShiny, the reactive source is known as the reactive value, the reactive endpoint is the observer, and the reactive conductor is the reactive expression. There are two fundamental differences between the observer and reactive conductor's functionality. The first is that the observer responds to event flushing but reactive expressions don't. The second is that reactive expressions can return values but the observer can't.

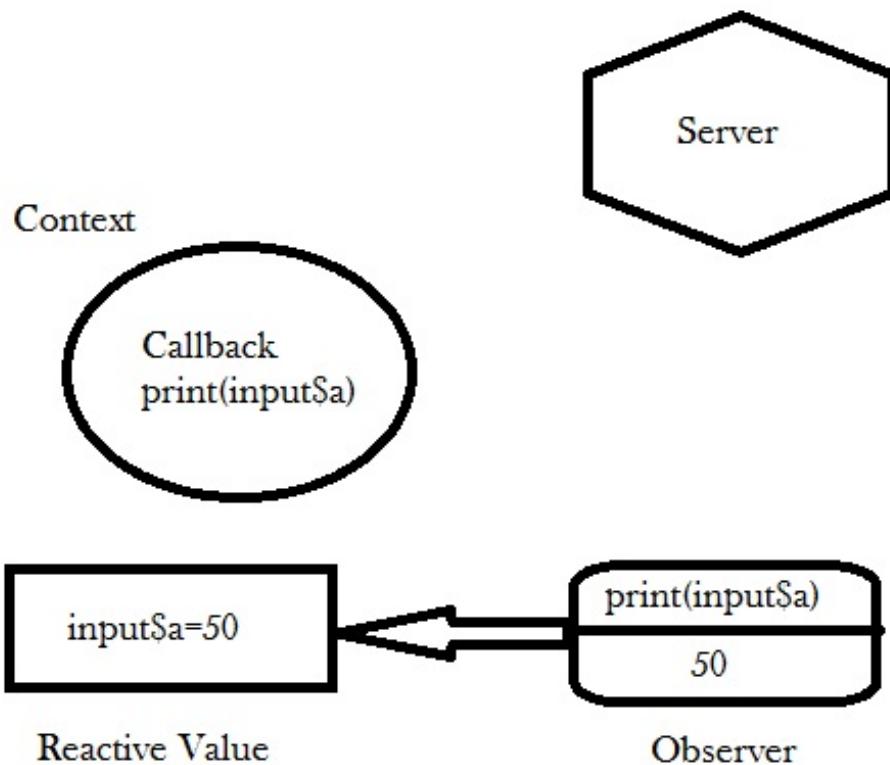
A closer look at reactivity

So far, we have discussed reactive programming and how Shiny implements its features. We'll now take a closer look at how this works. RShiny uses a pull mechanism instead of a push mechanism. This means whenever the output senses that there is a change in input, it pulls the new values from the input. In push mechanisms, output values are pushed for every input change. Pull mechanisms are also known as lazy-evaluation mechanisms. This simply means that they are not like input to output electricity transfers, but more like pigeon-carrier methods.

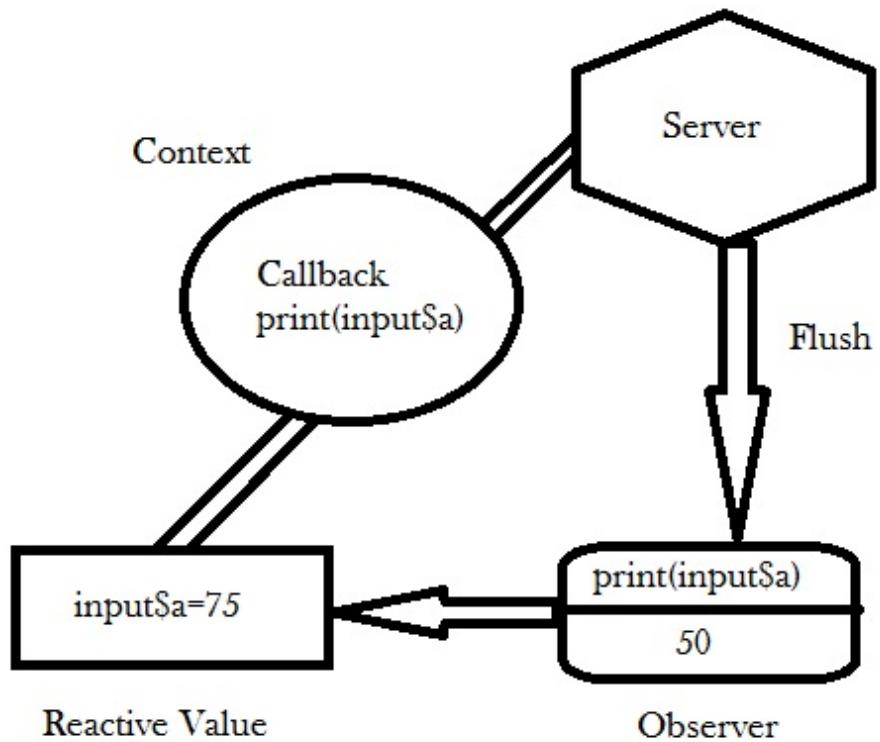
Shiny implements reactivity with two special object classes:

- Reactive values, which, as we have seen, can be printed as `input$value`
- Observer, which can be printed as `output$output_val`

Whenever an observer uses a reactive value, it creates a reactive context with the value. The context can be any expression that is run if there is any change in the value. Here, the expression refers to callback. If multiple observers are using the same value, that single value can hold multiple contexts:



From the preceding diagram, see how context works by looking at the basic communication between reactive values and observers. Suppose that the input value changes to `60`, so we get `input$a=60`. In this case, a context with a new value will be created. The observer senses that new value and asks the server for callback. The server finds the context with the new value and flushes it to the observer:



In a graphical representation, the reactive source can only be a parent and the source end can only be a child, whereas a conductor can be a parent and a child.

Controlling specific input with the isolate() function

There may be situations when the observer or conductor wants to read the value of an expression but avoid dependency. Suppose we want to import some data or perform some calculations, but only after a button has been clicked.

Let's take a look at this example:

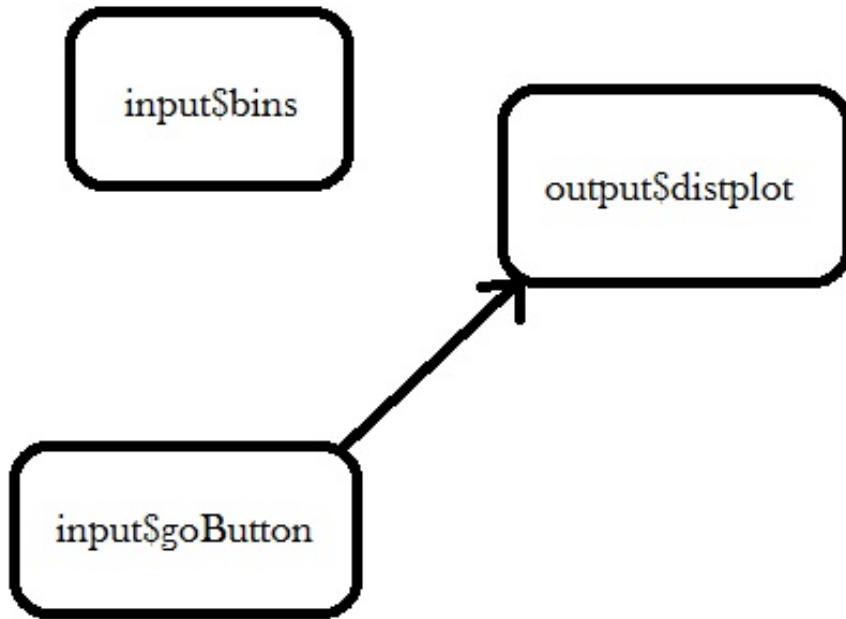
```
library(shiny)
# Define UI for application that draws a histogram
ui<- fluidPage(
  # Application title
  titlePanel("Old Faithful Geyser Data"),
  # Sidebar with a slider input for number of bins
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                 "Number of bins:",
                 min = 1,
                 max = 50,
                 value = 30)
    ),
    # Show a plot of the generated distribution
    mainPanel(
      actionButton("goButton", "Go!"),
      plotOutput("distPlot")
    )
  )
)
# Define server logic required to draw a histogram
```

In the server file, we can see that `output$distPlot` is dependent on `input$goButton`. Whenever the button is clicked, the plot gets executed. However, when we wrap `input$bins` in `isolate()`, this tells Shiny that the observer or reactive expression should not be dependent on any reactive object:

```
server<- function(input, output) {
  output$distPlot<- renderPlot({
    # generate bins based on input$bins from ui.R
    x <- faithful[, 2]
    # bins<- seq(min(x), max(x), length.out = input$bins + 1)
    # Take a dependency on input$goButton
    input$goButton
    dist<- isolate( bins <- seq(min(x), max(x), length.out = input$bins + 1))
    hist(dist, breaks = bins, col = 'darkgray', border = 'white')
    # draw the histogram with the specified number of bins
    # hist(x, breaks = bins, col = 'darkgray', border = 'white')
```

```
|   })  
| }  
# Run the application  
shinyApp(ui = ui, server = server)
```

The flow graph looks as follows:



In the code, we can prevent the plot being shown the first time without the click button being pressed by applying conditions to it. In `isolate()`, not only reactive but also reactive expressions can be included. It is also possible to include multiple statements in an isolate block.

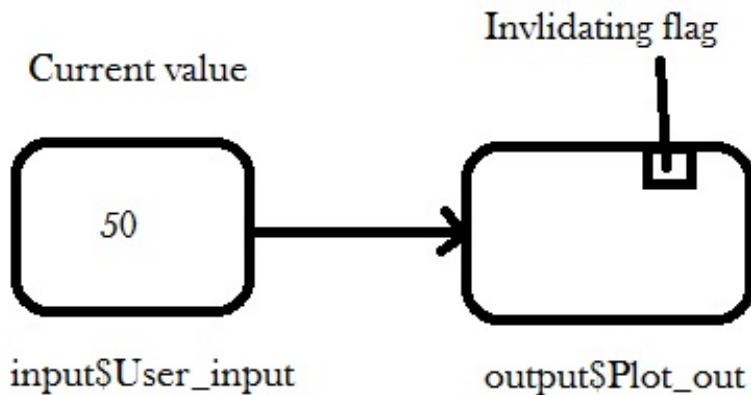
Running reactive functions over time (execution scheduling)

The core of the Shiny application is the reactive engine, which tells Shiny when to execute each component. We are now familiar with the reactive flow of various components. Reactive observers have a flag that indicates whether they have been invalidated. Whenever the values of the input object change, all of the descendants in that graph are invalidated. Such invalidated observers are also called dirty or clean. Along with this, the arrows in the flow diagram that have been followed are removed.

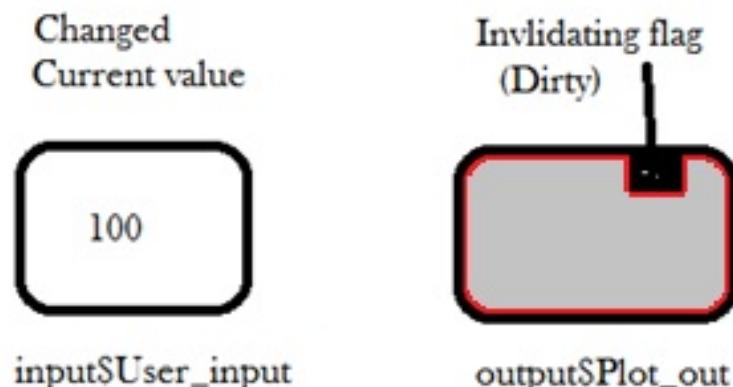
Let's discuss an example of a single reactive source and endpoint:

```
server<- function(input, output) {  
  output$Plot_output<- renderPlot({  
    hist(rnorm(input$user_input))  
  })  
}
```

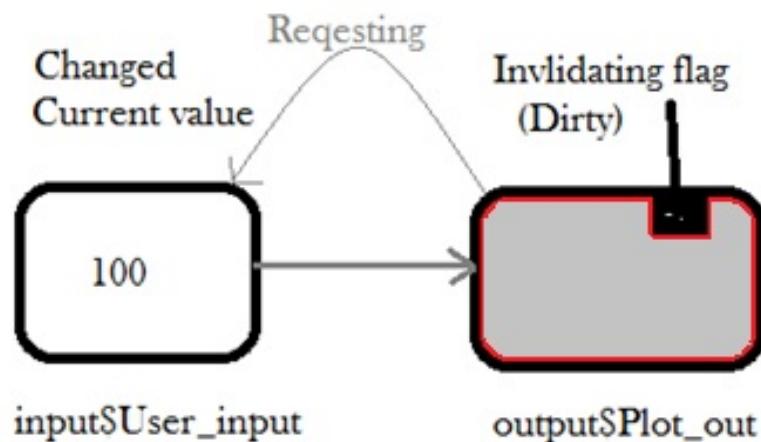
The flow diagram is as follows:



As soon as the input values change, all the descendants are invalidated and a flush event is triggered:

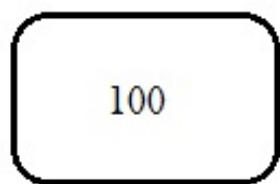


This is represented in the following diagram, in which the output object is shown in dark grey. When this happens, all the invalidated observers are re-executed. If the output object re-executes, it accesses the reactive value. This makes the output object dependent on the input:

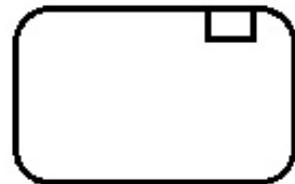


From the preceding graph, we can see that `output$Plot_out` is requesting for input. Once it gets input, the invalidating flag is cleared and the output is placed on the UI:

Output
ready



input\$User_input



output\$Plot_out

Event-handling using observeEvent and eventReactive

So far, we have looked at how reactivity works in Shiny applications. We have learned how to use `observe()` and `isolate()`. We will now look in more detail at event-handling. An event can be defined as a reactive value or an expression that triggers other calculations. For example, sometimes we want some actions to happen only after the action button is clicked.

We have already learned to handle events using `observe()` and `isolate()`. There are two more methods:

- `observeEvent`
- `eventReactive`

These two provide straightforward ways of handling events:

- `observeEvent`: If we want to perform an action in response to an event, `observeEvent` is useful. The syntax is as follows:

```
observeEvent(eventExpr, handlerExpr, event.env = parent.frame(),
event.quoted = FALSE, handler.env = parent.frame(),
handler.quoted = FALSE, label = NULL, suspended = FALSE, priority =
0,
domain = getDefaultReactiveDomain(), autoDestroy = TRUE,
ignoreNULL = TRUE, ignoreInit = FALSE, once = FALSE)
```

The first argument is the event to be responded to and the second is the function to be called when the event occurs. The rest of the parameters are optional and an explanation can be seen in next section of `eventReactive`.

- `eventReactive`: `eventReactive` is used to calculate a value that only gets updated when a response to an event is needed. The syntax is given here:

```
eventReactive(eventExpr, valueExpr, event.env = parent.frame(),
event.quoted = FALSE, value.env = parent.frame(), value.quoted =
FALSE,
label = NULL, domain = getDefaultReactiveDomain(), ignoreNULL =
```

```
| TRUE,  
| ignoreInit = FALSE)
```

Here, the first argument is the event to be detected and the second is the value expression. The rest of the arguments are explained here:

- `event.env`: The parent environment for the event expression. By default, the parent environment is the calling environment.
- `event.quoted`: A logical expression to tell whether `eventExpr` is quoted.
- `handler.env` and `handler.quoted`: The same as `event.env` and `event.quoted`, with respect to the handler.
- The label is the name given to the observer or the reactive.
- Suspended: A logical expression for identifying whether the observer is suspended.
- Priority: A value that identifies the priority of the observer.
- `autoDestroy`: Another logical expression. If it is true, the observer will be destroyed after the domain has ended.
- `ignoreNULL`: A logical parameter as well. If it is true, the action is to be triggered for the value is NULL.
- `ignoreInit`: False by default. If it is `TRUE`, and `observeEvent` is first created or initialized, `handlerExpr` (second argument) is ignored.
- `valueExpr`: An expression that returns the value of `eventReactive`.
- `value.envs`: The parent environment for `valueExpr`.
- `value.quoted`: Checks whether `valueExpr` is quoted or not.

`library(shiny):`

```
| shinyApp(  
|   ui = fluidPage(  
|     column(4,  
|       numericInput("x", "Value", 1),  
|       actionButton("button", "Show")  
|     ),  
|     column(8, tableOutput("table"))  
|   ),
```

In the preceding code, there is one input box and an action button on the UI. The following is the code of the `server` section. Here, we can see that `observeEvent` is handling the event, showing the input values, and `eventReactive` is calculating the head rows from the `iris` datasets with a given value of `x`:

```
| server = function(input, output) {  
|   observeEvent(input$button, {  
|     cat("Showing", input$x, "rows\n")  
|   })
```

```
| df<- eventReactive(input$button, {  
|   head(iris, input$x)  
| })  
| output$table<- renderTable({  
|   df()  
| })  
| })
```

You will get the following output:

The screenshot shows a Shiny application window titled "G/Packt/Author/Evaluated/version_control - Shiny". The URL is "http://127.0.0.1:5457". The window contains a table with the following data:

| Value | Sepal.Length | Sepal.Width | Petal.Length | Petal.Width | Species |
|-------|--------------|-------------|--------------|-------------|---------|
| 1 | 5.10 | 3.50 | 1.40 | 0.20 | setosa |

Below the table is a "Show" button. The window has standard operating system window controls (minimize, maximize, close) and a "Publish" button. At the bottom, there is a taskbar with various icons and the system clock showing "10:44 AM 15-09-2018".

Functions and modules

We have now gone through a good number of Shiny applications that are focused on different topics. Imagine that we now have to develop a fully-fledged enterprise application with various functionalities as a Shiny application. As the application grows, so will the number of lines of code and its complexity. In situations such as these, applications can become unmanageable. We may also be rewriting the same kind of code repeatedly, thereby increasing the development time. Additionally, variables need to be scoped, which limits the reusability of names. To eliminate these obstacles, we can use modularization.

Modularization is nothing but the use of separating modules according to their functionality. Modules can be defined as a set of instructions that are written to accomplish a task. R programming is developed around modules. We repeatedly call functions for almost any task we want to perform. This means that we don't have to write more code, as is required in programming languages such as C, C++, or Java. We can use this programming paradigm with Shiny as well.

So far, we have coded UI and server functions in the Shiny application, with a number of components and input and output elements. We will now try to modify these in a simpler coding pattern with modules.

We can organize the code into two sections, based on two functions:

- For creating UI elements
- For loading server logic

We also have to stick to some naming conventions so that the code can be identified easily. The UI modules can be suffixed with Input, Output, or UI. Before going into too much detail, let's start by observing the simple application code that is available whenever we start the Shiny application, RStudio. This is given here:

```
#  
# This is a Shiny web application. You can run the application by clicking  
# the 'Run App' button above.  
#  
# Find out more about building applications with Shiny here:
```

```

#      http://shiny.rstudio.com/
#
library(shiny)
# Define UI for application that draws a histogram
ui<- fluidPage(
  # Application title
  titlePanel("Old Faithful Geyser Data"),
  # Sidebar with a slider input for number of bins
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                 "Number of bins:",
                 min = 1,
                 max = 50,
                 value = 30)
    ),
    # Show a plot of the generated distribution
    mainPanel(
      plotOutput("distPlot")
    )
  )
)

# Define server logic required to draw a histogram
server<- function(input, output) {

  output$distPlot<- renderPlot({
    # generate bins based on input$bins from ui.R
    x <- faithful[, 2]
    bins<- seq(min(x), max(x), length.out = input$bins + 1)

    # draw the histogram with the specified number of bins
    hist(x, breaks = bins, col = 'darkgray', border = 'white')
  })
}

# Run the application
shinyApp(ui = ui, server = server)

```

We can see that this code from RStudio is simply written without modularization. Let's discuss in detail how we can modularize the UI and Server code of this application:

- **UI Code modularization:** In the preceding code, one of the elements of the UI section is `sliderInput`. At the moment, there is only one `sliderInput` box, but there might be more, meaning we would have to write the same code again and again if we didn't use modularization. Let's develop a module for this. For the UI module, we need to give a name to the function, such as `sliderbarInput`. We also need to give an ID to each input element that we are using. In the function argument, therefore, we will add one `ID` parameter, which will be used to create a namespace using `ns()`. Right now,

we only have one input element, but in the future we may want to add multiple elements. We will use `tagList()`. The UI module will now look as follows:

```
SliderbarInput<-function(id){  
  ns<-NS(id)  
  tagList(  
    sliderInput(ns("bins"),  
               "Number of bins:",  
               min = 1,  
               max = 50,  
               value = 30)  
  )  
}
```

We have now separated the input code from the UI code. We have to call this code from inside the UI whenever we want to put in a slider bar. In the following code, we can see that `SliderbarInput("Simplesliderbar")` has been called. This will call to the preceding code. "Simplesliderbar" is the ID given to `sliderInput`:

```
ui<- fluidPage(  
  # Application title  
  titlePanel("Old Faithful Geyser Data"),  
  # Sidebar with a slider input for number of bins  
  sidebarLayout(  
    sidebarPanel(  
      SliderbarInput("Simplesliderbar") # call to function  
    ),  
    # Show a plot of the generated distribution  
    mainPanel(  
      plotOutput("distPlot")  
    )  
  )  
)
```

- Server Code modularization: In the server section of the original code, we are developing a calculation to render a plot. We will develop a module to perform this task. Our first task is to give a name to the function, such as `Sliderbar`. In the server module, the function has to have three arguments: `Input`, `output`, and `session`. The rest can be the input parameters the user wants to add. After that, we will write all the calculation code in this function. Our new module will look as follows:

```

Sliderbar<-function(input,output,session){
  # generate bins based on input$bins from ui.R
  x      <- faithful[, 2]
  bins<- seq(min(x), max(x), length.out = input$bins + 1)

  # draw the histogram with the specified number of bins
  hist(x, breaks = bins, col = 'darkgray', border = 'white')
}

```

We now have to call the preceding code from the `usingcallModule()` server code. Here, we will pass the function to be called and the label or ID of the input element. The server code will look as follows:

```

server<- function(input, output) {
  output$distPlot<- renderPlot({
    callModule(Sliderbar, "Simplesliderbar")
    # Simplesliderbar is coming from UI
  })
}

```

By doing this, we have recreated the application. However, we have to decide where we can define our modules. One option is to keep our modules in the preamble of a single file application or in a file that is sourced in the preamble of a single file. Another option is to put the modules in a `global.r` file, or a file that is sourced in the global file. Alternatively, we can wrap a module in a package that is loaded by the application.

The advantage of using modularization is that it makes the code reusable.

Shinytest

Let's say that the Shiny application we have developed runs well on our machine for some input, but for others it does not give the desired output or it gets stuck somewhere and throws errors. In the software development process, testing is one of the most important tasks. A Shiny application might stop working due to any of the following reasons:

- The version of Shiny and the version of the packages may differ
- A modification in the application code leads to the wrong input for any other reactive code
- The data format may have changed

There may be countless reasons that cause us to come across errors and cause the application to stop working. Doing testing manually can be time-consuming and inefficient because you have to consider a wide range of use cases. For this reason, Shiny has the `shinytest` package for automatic testing. It can be installed as follows (https://www.rdocumentation.org/packages/devtools/versions/1.13.6/topics/install_github):

```
| library(devtools)
| install_github("rstudio/shinytest")
```

To carry out a test, follow these steps:

1. Run `recordTest()` to launch the app in a test recorder (<https://rstudio.github.io/shinytest/reference/recordTest.html>):

```
| library(shinytest)
|
| # Launch the target app (replace with the correct path)
| recordTest("path/to/app")
```

In the test recorder, we can find a list of recorded events. These events are interactions made by the user. We can also take snapshots of the state of the app by clicking on `take snap` on the right-hand side window of the recorder.

2. Quit the test recorder and find the test script in the `.R` file in the `test/`

subdirectory. This holds code like the following:

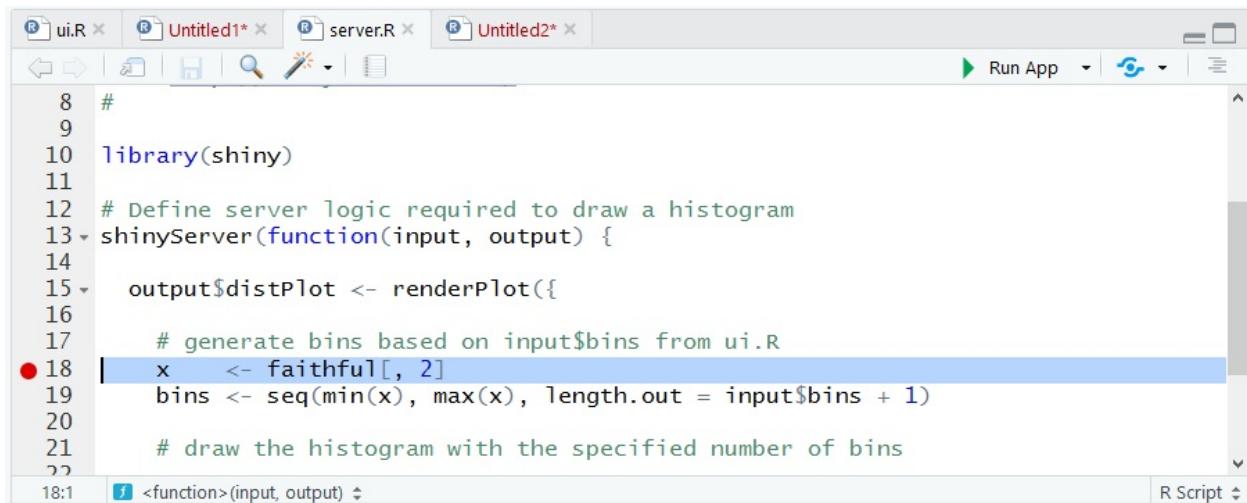
```
app<- ShinyDriver$new("..")
app$snapshotInit("mytest")
app$snapshot()
app$setInputs(checkGroup = c("1", "2"))
app$setInputs(checkGroup = c("1", "2", "3"))
app$setInputs(action = "click")
app$snapshot()
```

Here, we try the possible use case input and figuring errors. For a more in-depth understanding of the workings of the `shinytest` package, refer to <https://rstudio.git-hub.io/shinytest/articles/shinytest.html>.

Debugging

Debugging a Shiny app is not the same as debugging code in other programming languages, such as C, C++, or Java, where the control flow looks linear. Because RShiny is reactive, it runs on a web server as well as a Shiny framework. That makes it harder to debug.

If we get an error or an undesirable output, we can apply break points at the suspected line of code. This can be done by clicking on the left side of the code where the line number is given, which makes a red dot appear:



The screenshot shows the RStudio interface with the 'server.R' tab active. The code editor displays the following R script:

```
8 #  
9  
10 library(shiny)  
11  
12 # Define server logic required to draw a histogram  
13 shinyServer(function(input, output) {  
14  
15   output$distPlot <- renderPlot({  
16  
17     # generate bins based on input$bins from ui.R  
18   | x <- faithful[, 2]  
19     bins <- seq(min(x), max(x), length.out = input$bins + 1)  
20  
21   # draw the histogram with the specified number of bins  
22 } )  
18:1 | <function>(input, output) ▾
```

A red dot is visible on the left margin next to the line number 18, indicating a breakpoint has been set. The line 18 contains the assignment `x <- faithful[, 2]`.

After running the code, Shiny will stop execution at the breakpoint and we can step into the code and take a look at the current variable values. Setting a breakpoint is possible with RStudio.

Sometimes, applying a breakpoint doesn't work, so we have to change the input and observe the output. We can then apply a break point again to try to diagnose the problem. We can enable the `showcase` mode and see which part of the code is executing:

```
| shiny::runApp(display.mode="showcase")
```

To examine the reactive, we can enable `shiny.reactlog` and observe the reactive logs:

```
| options(shiny.reactlog=TRUE)
```

This will give us an idea about how the reactives are working. For more detailed debugging, we can apply a print statement in suspected places to understand the flow of the application and the behavior of the code. In order to trace the level of the client or server architecture, we need to set the `shiny.trace` mode to on. This will provide a trace of the app in the JSON file.

The `browser()` statement can also be used for debugging. It acts as breakpoint and can be added anywhere in the code.

Debugging with the preceding methodologies is sufficient to run an application, but there are many other methods, which are outside the scope of this book. After debugging, the next step is to handle the errors that are detected.

Handling errors (including validate() and req())

Are you getting an error message in red on your screen? Are you or your user able to understand that message? The answers to such questions can be validation errors using `validate()` and `req()`.

Validate

This is designed to lead the user through the UI of the Shiny app. Imagine that we have a situation where we need to plot a graph based on some values of a list box and it throws up a red message. This looks stressful to the user. We need to provide a precautionary message to make sure the reader enters a valid input. `validate` is a way of checking the input and providing a message, using `need`. In the following code, we can see that validate is checking for a phone number. If the input is an empty string, it will give a message:

```
server<- function(input, output) {  
  Phon_Number<- reactive({  
    validate(  
      need(input$PN != "", "Provide Phone number")  
    )  
    get(input$data, 'package:datasets')  
  })
```

Handling missing input with req()

Sometimes, we might see an error when our graph needs to be updated dynamically because the data is unavailable while the application is loading. It is also possible that the data is not available for a certain input and the element tries to render the output. In this situation, we would see an error message.

To handle this, we can use `req()`. `req()` is the short form of require. This is useful for checking preconditions. Whenever we need to write a reactive element, we can enclose it in `req()`. This will make sure that the output is halted until the values are available.

Profiling R code

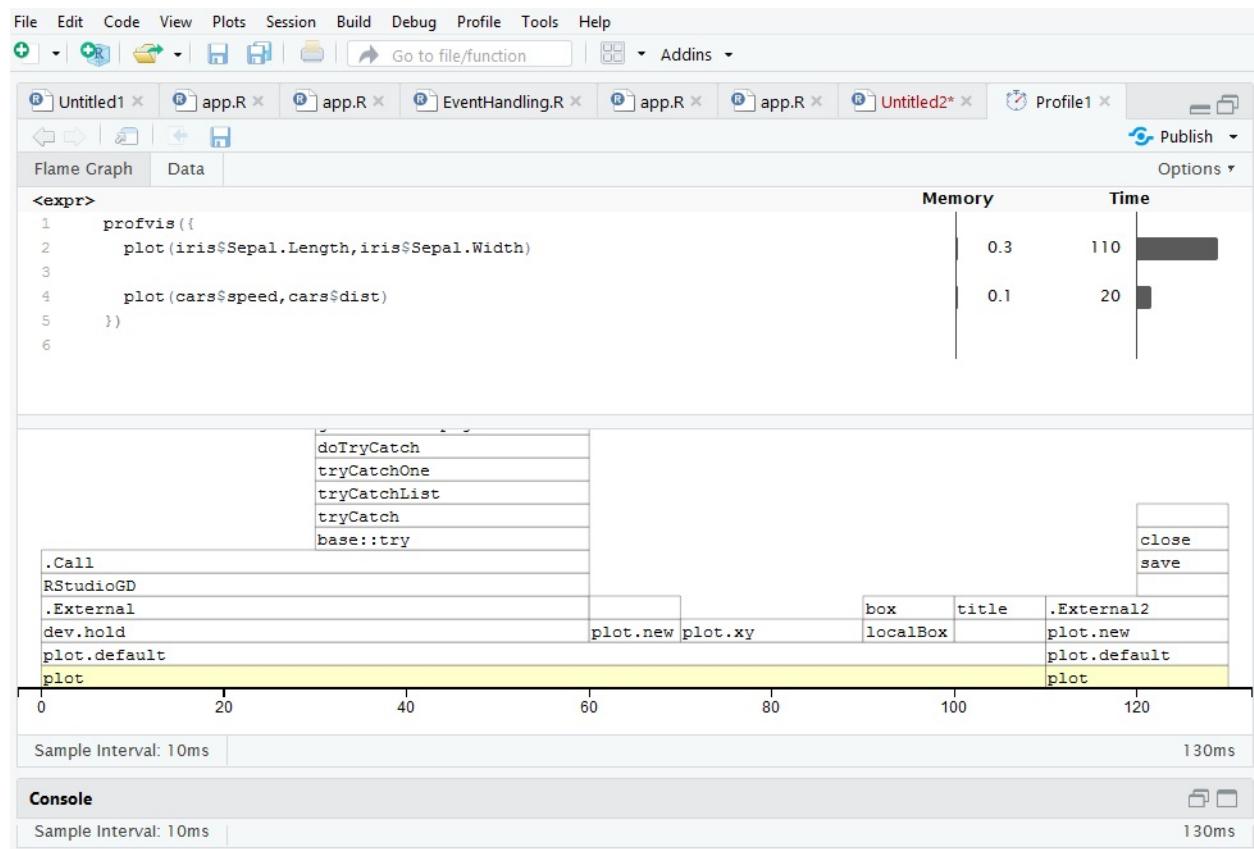
In the big data era, where we might have petabytes of data, maintaining the performance of applications is crucial. The application performance might go down because of data-loading operations or calculations. To detect such operations, we have a package called `profvis`. This tells us how much time a line of code takes to execute. Install the package as follows:

```
| install.packages("profvis")
```

RStudio has provided support for the `profvis` package. Let's take a look at how to use it:

```
library(profvis)
profvis({
  plot(iris$Sepal.Length,iris$Sepal.Width)
  plot(cars$speed,cars$dist)
})
```

`profvis` encloses the code to be analyzed. In the following screenshot, we can see the output. For each line of code, we can see the memory usage with the time taken:



Debounce and throttle

Debounce and **throttle** are used to slow down a reactive expression. For example, suppose we are using the invalidation check for a reactive expression and error indications are prompted unnecessarily. We can use debounce and throttle to make expressions such as these slow down and wait for intermediate expressions to complete their calculations. The syntaxes of both of these are as follows:

```
| debounce(r, millis, priority = 100, domain = getDefaultReactiveDomain())
| throttle(r, millis, priority = 100, domain = getDefaultReactiveDomain())
```

Here, `r` is the reactive expression that invalidates too often. `millis` is the time window used by `debounce/throttle`, and `priority` sets the observer's priority. For example, if we want to add `debounce` to an expression, we can do it as follows:

```
plot_iris<- plot(iris$Sepal.Length,iris$Sepal.Width) ) %>% debounce(1000)
For more detail visite https://shiny.rstudio.com/reference/shiny/1.0.0/debounce.html. Lets have an exam

## Only run examples in interactive R sessions
if (interactive()) {
  options(device.ask.default = FALSE)

  library(shiny)
  library(magrittr)

  ui <- fluidPage(
    plotOutput("plot", click = clickOpts("hover")),
    helpText("Quickly click on the plot above, while watching the result table below:"),
    tableOutput("result")
  )

  server <- function(input, output, session) {
    hover <- reactive({
      if (is.null(input$hover))
        list(x = NA, y = NA)
      else
        input$hover
    })
    hover_d <- hover %>% debounce(1000)
    hover_t <- hover %>% throttle(1000)

    output$plot <- renderPlot({
      plot(iris)
    })

    output$result <- renderTable({
      data.frame(
        mode = c("raw", "throttle", "debounce"),
        x = c(hover_d$x, hover_t$x, hover_d$x),
        y = c(hover_d$y, hover_t$y, NA)
      )
    })
  }
}
```

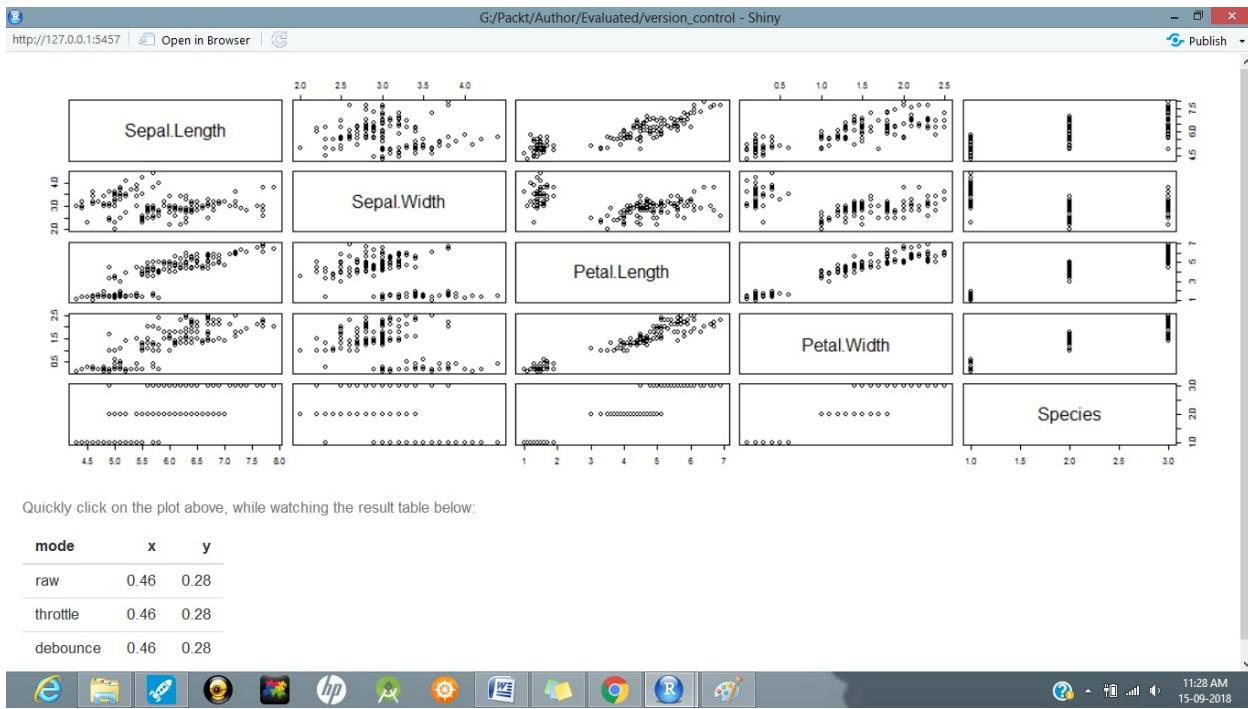
```

        y = c(hover()$y, hover_t()$y, hover_d()$y)
    })
}

shinyApp(ui, server)
}

```

You will get the following output:



Quickly click on the plot above, while watching the result table below:

| mode | x | y |
|----------|------|------|
| raw | 0.46 | 0.28 |
| throttle | 0.46 | 0.28 |
| debounce | 0.46 | 0.28 |

Result table

Summary

In this chapter, we learned about the reactivity of RShiny. We learned in detail how reactive source, endpoint, and conductor work, and how to control specific input with `isolate()`. We have also scheduled the reactive of functions. After that, we used `observeEvent` and `eventReactive` for event-handling, and learned how to make our code more modular. After developing our application, we carried out testing using `shinytest`, debugging, and error-handling. On top of all this, we discussed how to improve the performance of our Shiny code using profiling.

Persistent Storage and Sharing Shiny Applications

Having made all of those wonderfully intuitive and powerful applications, you are quite naturally going to want to show them off. You may wish to share them with colleagues or members of the worldwide R community. You may wish to share them with individuals in your department or field who, while not R users, can handle a little bit of effort to get an application working. Or you may wish to share them transparently and freely with the whole world by hosting them on a server. Shiny offers quite a lot of approaches to sharing applications, and you'll be glad to hear that even the most complex should not be too taxing, with the right hardware and OS on your server.

In this chapter, we will take a look at the following:

- Sharing over GitHub
- An introduction to Git using Git and GitHub within RStudio
- Sharing applications using Git
- Sharing using `.zip` and `.tar`
- `Shinyapps.io`
- Shiny Server
- Running Shiny in AWS and Google Cloud
- Scoping, loading, and reusing data in Shiny applications
- Temporary data input/output
- Permanent data functions
- Databases
- SQL injection
- Databases with the pool package

There are a few ways of sharing with R users running the Shiny package within R summarized in the following sections.

Sharing over GitHub

By far, the easiest way to share your creations with fellow R users is over GitHub (github.com). Of course, other R users can also use all the other methods in this chapter, but this is probably the most frictionless method (short of hosting the application) for both you and the end user.

An introduction to Git

You will no doubt have heard of Git (git-scm.com)—the version-control system that has collaborative sharing features at GitHub), even if you have never used it. Git is a version-control system that can be used locally on your computer, or in order to get the best out of it, the version-control repository on your computer can be synced online at GitHub. Hosting of open source code at GitHub is free, and there are paid options for closed source code. If you haven't already used a version control, this is an excellent reason to start. It is a little intimidating for newcomers, but over time, the resources and tutorials on the site have improved and perhaps one day of head-scratching awaits you. Trust me that one day I will be paid back hundredfold. The Pro Git book can be downloaded for free from the Git site at git-scm.com/book/en/v2. There is also a wonderful interactive tutorial (try.github.io) on the Git site. As a die-hard Linux enthusiast, it pains me to admit it, but I actually found learning on Windows easier because they provide a wonderful GUI to get you started (also on OS X). This does not mean that you need to use Windows or should stick to Windows; I happily dropped the GUI and went to the terminal in Linux once I'd found my feet a bit.



It's also worth noting that there are some great GUIs for Linux as well, so you can check your package-management system. I didn't find any that supported beginners so well as the official Windows or OS X versions, though. Git has a list of GUIs at git-scm.com/downloads/guis. Note that some of these support GitHub and others support Git itself. The list includes the Windows, OS X, and Linux GUIs.

Finally, RStudio itself actually supports Git and GitHub, and once you've installed Git and set up your account, you can pretty much run the whole show from within RStudio itself.

Using Git and GitHub within Rstudio

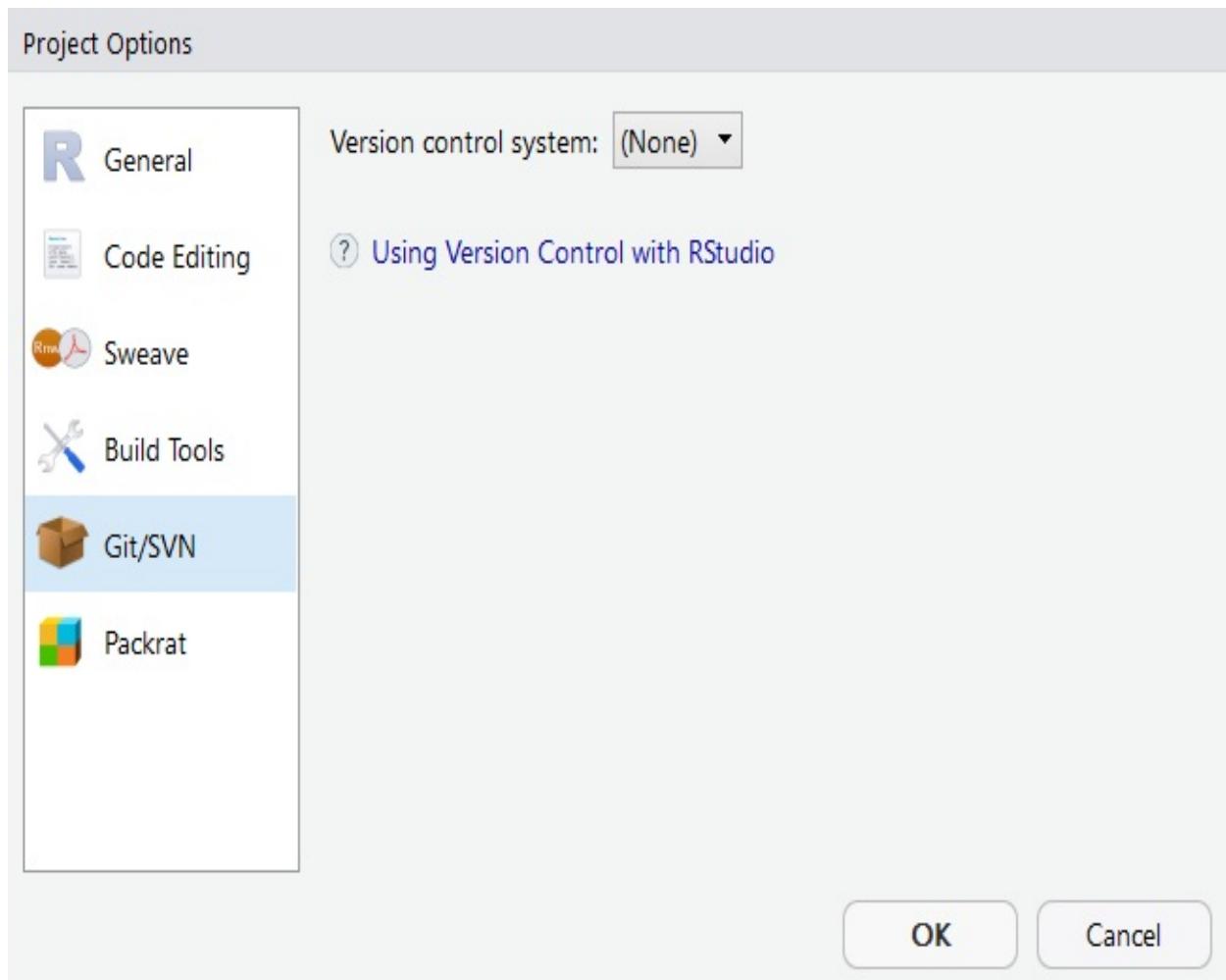
To install Git, simply go to the URL mentioned earlier and download the .exe file for Windows, or on Ubuntu, run the following command:

```
| sudo apt-get install git
```

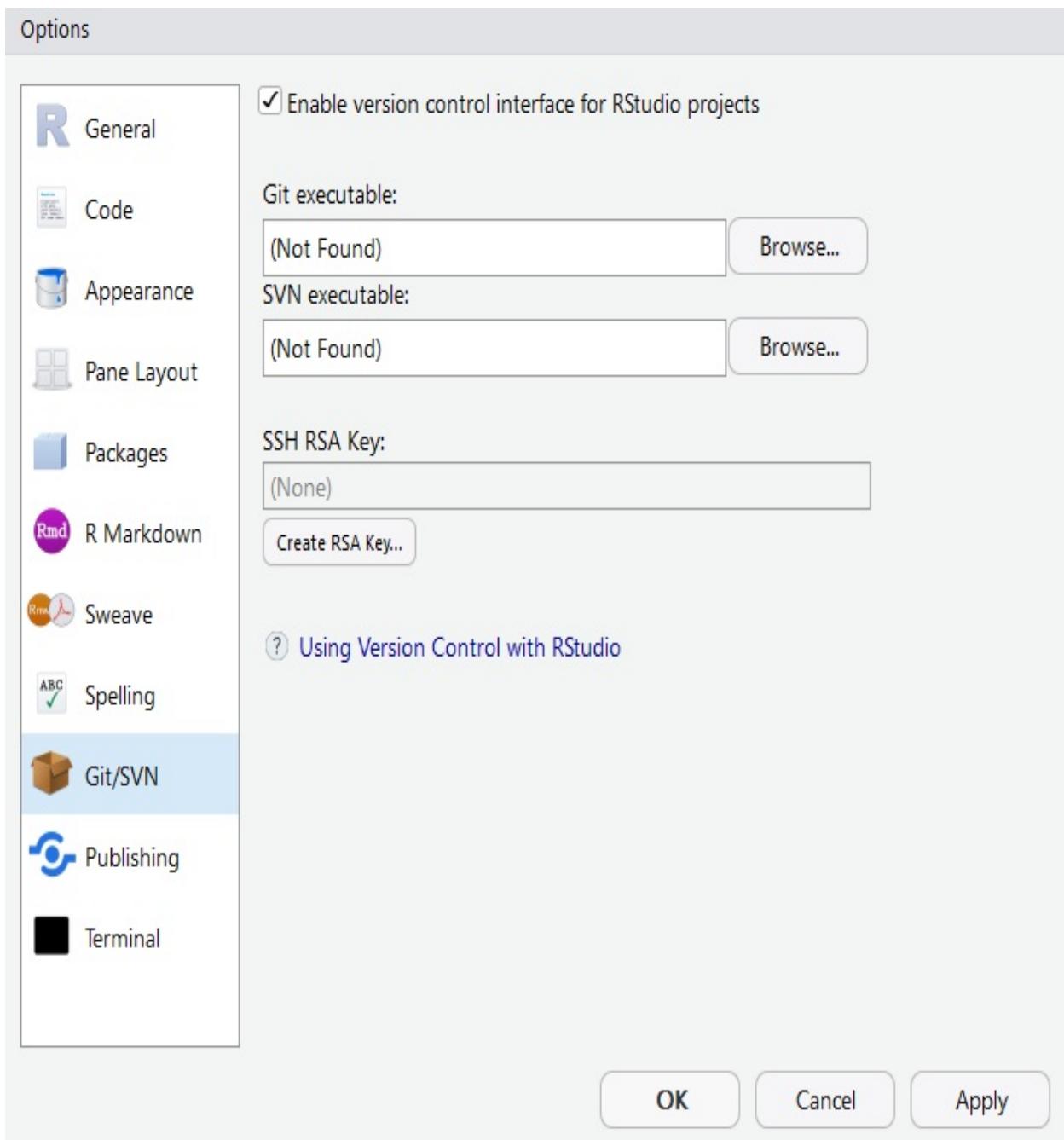
For other flavors of Linux, check the package-management system. Having installed Git, you now need to set up a new project within RStudio. A version-control with Git (or SVN, a different version-control system, which we will not consider here) is only possible when we use a project within RStudio.

Projects in RStudio (h3)

Using projects in RStudio is a good way to organize your work. Each project has its own working directory with a separate R session, workspace, console input history, and open editor tabs (among other things). Each time a project is opened, each of these will be set to the value currently associated with the project, in effect launching a new R session, loading the data and console history since the last time the project was used (if they are selected as the default behavior or individually for this project), setting the working directory to the one associated with the project, and so on. This allows you to switch in and out of different projects either as you work or when you pick up work the next day. To set up a new project, go to File | New Project in RStudio. Select either New Directory if this is a completely new set of code and files that you want to create a new folder for, or Existing Directory if you have already started and just want to point the project to a directory that you have already created. Once you have a project set up, go to Tools | Version control | Project Setup.... The following menu will appear:

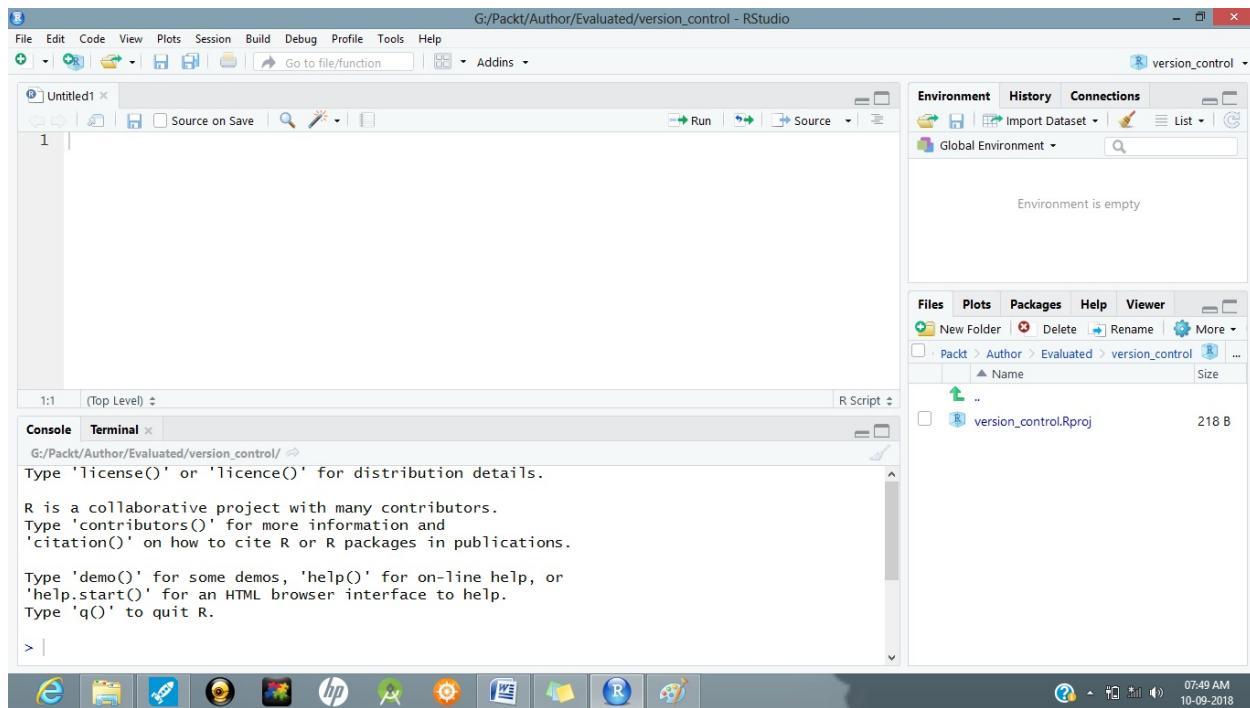


Make sure that the Git/SVN tab on the left-hand side of the page is selected and use the version-control system control on the right-hand side of the page to select Git/SVN, if you prefer, but this will only appear if you have installed it, and this will not be covered in this chapter. You may need to reopen the project at this point by going to File | Recent projects. You will need to configure the remote connection between your local `.git` repository and the GitHub account yourself. Go to your GitHub account, and go to Repositories | New. Give it a name and description, and select Create repository. Having done this, some instructions will appear on the screen that will help you to set up a connection between this remote repository and the local version on your machine. At the time of writing, the simplest way of doing this is the third box down. Keep these instructions as you will need them later, but for now, we need to configure RStudio a little further. Go to the Tools | Global options, and select the Git/SVN tab. The following menu will appear:



Check whether the Git executable is set up correctly in the first line. If you already have an SSH RSA key, it should be displayed in the bottom line. If not, click on Create RSA Key, and you will be guided to create one. If you have not previously paired your RSA key with your GitHub account (which you would not have done if this is your first experience with GitHub), click on View public key above the third line, and then copy the resulting text into your GitHub account by going to your account settings. This can be achieved by clicking on

your user portrait at the top-right corner of the GitHub web page. Next, click on Settings, and then click on the left-hand side of the screen and select SSH keys, and finally, click on Add SSH key, paste your key, and click on Add key. Having done this, you will need to commit your code to Git, that is, to the local copy on your machine. This is very easy in RStudio. Select the Git tab in the Environment pane in RStudio (by default, it's the top-right tab on the screen), as shown in the following screenshot:



Select the elements that you want to commit by clicking on the boxes to the far left of the screenshot. This will be anything that you want to commit to Git for your first submission and anything that has changed for subsequent submissions. Click on Commit in the menu bar, which is visible in the screenshot. You will be prompted to review the changes in a new window as well as instructed to write a commit message in the top-right corner of this window. You cannot commit without a message. For your first commit, you might like to write `First commit of beta version`, and then for subsequent commits, you might make comments such as `Fixed jQuery bug` and `Added dashboard elements`, depending on the changes you have made. Finally, to push to GitHub for the first time, select More | Shell in the Git tab. This will open a terminal window. Remember the terminal commands that the GitHub web page gave us when we set up the new repository and the two-liner I told you to keep track of? This is where we need them. Line by line, copy

the two-liner from the web page. This will set up the connection between RStudio and GitHub. From now on, you can push your code (that is, upload it to GitHub) by committing and clicking on the Push button in the Git tab menu bar. This is a very brief introduction to Git, GitHub, and RStudio, and is designed to get you started. There is much more to learn about how to use these tools efficiently, and you are advised to take a look at the online documentation for all three in order to learn how to make this process even simpler and more powerful.

Sharing applications using Git

We need to consult the websites mentioned earlier for more details of each of these steps. Once you've set your Git version control and paired with an online repository at GitHub, you can very easily share your creations with anyone running the R and Shiny package using the `rungitHub()` command, which takes the name of the repository and the username as mandatory arguments:

```
| runGitHub("GoogleAnalytics2ndEdition", "ChrisBeeley")
```

Code and data are both automatically downloaded and run. If you are using RStudio and want to launch your own external browser, as opposed to using the one that is built-in, you need to add `launch.browser = TRUE`. If you don't want or need version control, and don't need data to be included in the download, a simpler option is to use Gist, which is also hosted at GitHub at gist.github.com. Using Gist is simply a matter of visiting the URL, setting up an account, pasting your code into it, and giving the `server.R` and `ui.R` files the correct filenames. You will then have a URL, using which you can show your code to others. Running this code from the Shiny package is just a matter of using `runGist()` with the URL or even using the unique numeric identifier from the URL.

`library(shiny):`

```
| runGist("https://gist.github.com/ChrisBeeley/a2f1d88dfedcd2e1cb59")
| runGist("a2f1d88dfedcd2e1cb59")
```

Sharing using .zip and .tar

Probably the next most frictionless method of distributing a Shiny application to R users is by hosting either a `.zip` or `.tar` file of your application, either over the web or FTP. You will need somewhere to host the file, and then users can run the application using `runUrl()`, as follows:

| `runUrl("http://www.myserver/shinyapps/myshinyapp.zip")`



Note that this URL is not real. You need to replace it with the address of your own file.

Of course, you can distribute a `.zip` file any way you like—your users need to only unzip and then use `runApp()` from within the directory, just as you do when testing the application. You can email the file and distribute it on a USB drive for any method that you choose. The disadvantages of this method are that your users have to unzip the file themselves (although this is unlikely to confuse many R users), and any changes made to the application will also need to be distributed manually.

Sharing with the world

In most cases, any serious work that you do with Shiny will at some point need to be shared with a non R-user, whether it's a nontechnical colleague in your department or the whole of the internet. In this case, a bit more of the legwork falls to you, but you should still be pleasantly surprised about how simple the process is. There are two options here: set up your own server or get a paid account with RStudio to do it for you.

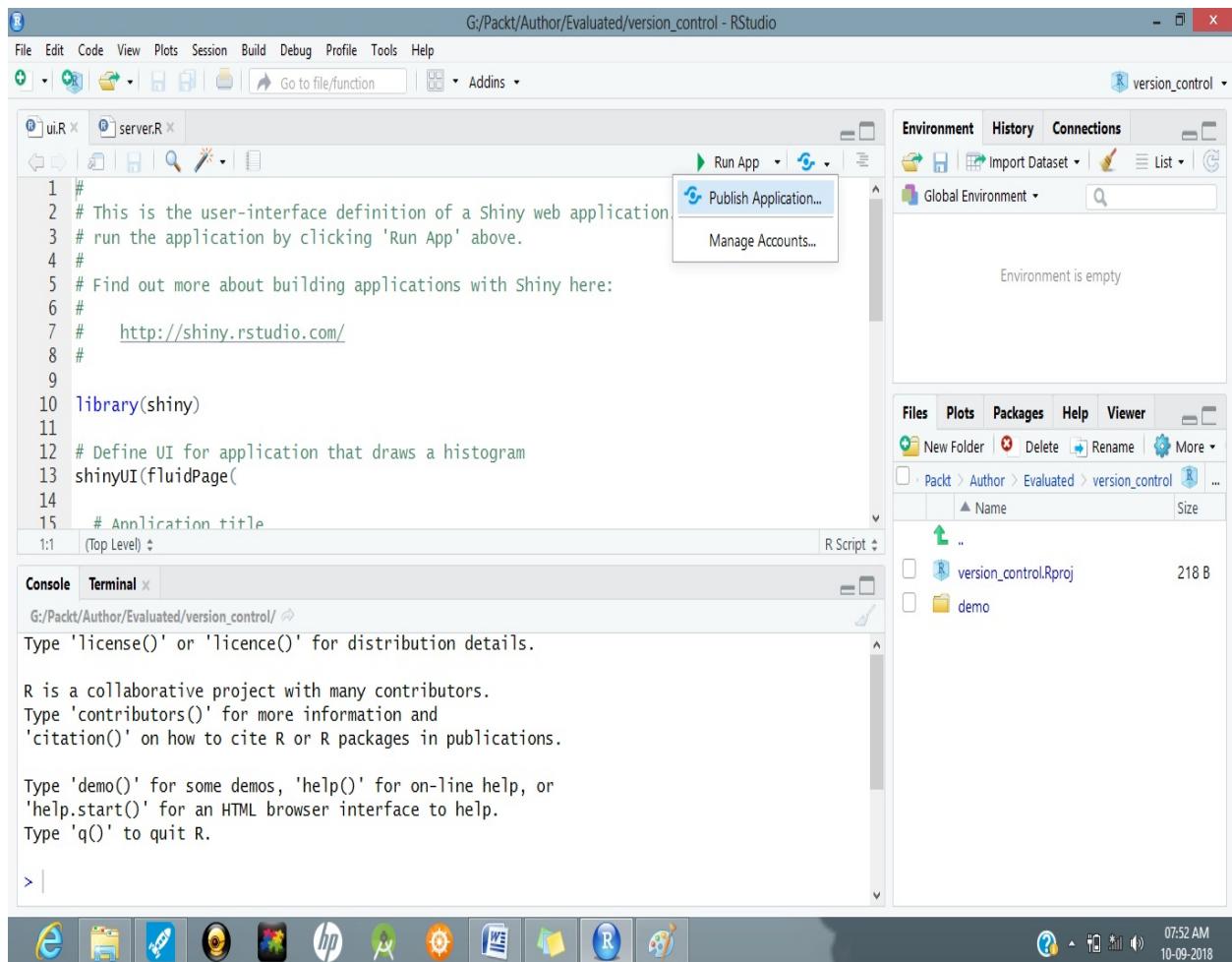
Shinyapps.io

`shinyapps.io` is RStudio's paid hosting for Shiny applications. At the time of writing, there is a tiered pricing structure, depending on the number of applications you wish to deploy, whether you need the authentication of users, the number of hours your applications will run per month, and so on. You can set up a free account that, at the time of writing, allows five applications and 25 hours of application runtime a month. This is welcome; however, it is only suitable for very small-scale use; a single tweet of your application on the `#rstats` hashtag is likely to bring enough traffic to your site to use all of the 25 hours in a very short time. Indeed, I have been linked to many `shinyapps.io` applications, which indicate that the account has exceeded the allocated hours this month and, therefore, does not work.



Be warned: if you want the world to see your application, you either need to get a paid account or run your own server (which is explained later). Using this service does, of course, entail copying your code and/or data to a third party, so if this is a problem for you, you will need to take a look at hosting yourself on a server.

If you are using RStudio, it is very easy to get an application on shinyapps. Whenever you have a Shiny application (that is, a `server.R` or `ui.R` file) open, you will find a little Publish icon in the upper-right corner of the editor, as shown in the following screenshot:



You will be prompted to install various things, depending on your OS and the configuration of your R installation. On Linux, you will probably save yourself a bit of configuration if you install the development version of R (`r-base-dev` on Ubuntu, available through the R metapackage on Fedora; for other distributions or operating systems, refer to the relevant documentation).



For all operating systems, you will be prompted to install various R packages. Users of Linux may have problems configuring some of these packages; you may need to install `libcurl-dev` and `openssl-dev` (or their equivalent for your distro). In Windows, in my experience, the whole operation, right from the vanilla installation of R, is completely seamless and everything will be installed and configured correctly.

Then, you will be prompted to go to your `shinyapps.io` account and log in, where you can authenticate RStudio. You can now publish it straight from RStudio:

1. Press the button highlighted in the preceding screenshot, select the files to be uploaded (for example, the code and data files), and click on Publish

2. It will launch a browser for you, so you can see the application for yourself and copy the link that needs to be shared
3. If you forget the link, just log into `shinyapps.io`—the link is available from your list of applications in the menu

Shinyapps.io without RStudio

It's not necessary to use RStudio to use `shinyapps.io`, it's just a bit easier. You need to follow these steps:

1. If you're happier in another IDE, you just need to ensure that you have the latest version of `devtools` installed:

```
|   install.packages('devtools')
```

2. Install shinyapps:

```
|   install_github('rstudio/shinyapps')
```

3. Load shinyapps:

```
|   library(shinyapps)
```

4. Log into your `shinyapps.io` account, copy the authorize token command from the tokens menu (token marked with xs here), and run it in your R session (note that this only has to be done once on each computer):

```
| shinyapps::setAccountInfo(name='chrisbeeley',
|   token='xxxxxxxxxxxxxxxxxxxxxx',
|   secret='xxxxxxxxxxxxxxxxxxxxxx')
```

5. Set your working directory to the folder that holds your application:

```
|   setwd("~/myShinyApp")
```

6. Deploy:

```
|   deployApp()
```

 More details are available on RStudio's pages at shiny.rstudio.com/articles/shinyapps.html.

Shiny server

If you want to host the applications yourself, Shiny Server is available for Linux. Again, there are paid and free options. Shiny Server is totally free and open source, which is a great credit to Rstudio. The paid version has a number of benefits. The main ones are the provision of support and extra features, particularly authentication (LDAP/PAM/Google accounts and running over SSL to encrypt data to and from the server). It also allows you to use multiple R processes to serve one application, supports multiple versions of R on the same server, and provides an admin dashboard that helps server administrators to monitor server load, concurrent connections, and so on.



Binaries are available for Ubuntu, Red Hat, CentOS, and SUSE Enterprise; for other distributions, it is possible to build from the source. The free version is, in my experience, stable and well-featured. Installation details can be found at rstudio.com/products/shiny/download-server/.

Follow the instructions mentioned previously to install, and using the default configuration, you should be able to navigate to a test Shiny application by going to `chrisbeeley.net:3838/shiny/01_hello/` in a web browser (replace the domain with your own URL). In order for Shiny Server to work, you need to open the relevant port (in this case, the default configuration, 3838) on your firewall. By default, applications are run from files located within `/srv/shiny-server`. You can include directories within this folder to organize your applications. The administrator's guide, which is linked to and from the download page, includes details of how to configure Shiny Server.

You may wish to change the port through which Shiny server communicates (again, opening this port on your firewall), change the location of application files, or add several locations to application files, or something else entirely. The complete details are available in the documentation. Installation on Ubuntu is embarrassingly easy; even with my limited knowledge of running Linux servers, I had it up and running on my personal server in less than an hour. It's run quite happily ever since. Mileage with other distributions will vary, although judging from forum and blog posts, people have successfully run it on a variety of distributions. Depending on what you are doing with your application, one thing

to be careful of is directory ownership and permissions.

For example, one of my applications produces PDF files for download. This requires making Shiny the owner of the directory within the application folder, which houses the temporary files that are produced for download and making the directory writable. In a corporate environment, you may also find that the port Shiny uses is blocked by the firewall—changing to a different port is simply a matter of editing the configuration file as detailed on the Shiny server web page given previously. If you are in a corporate environment running Windows, it's worth noting that the open version runs fine on an Ubuntu server virtualized on Windows, in my experience. I couldn't speak for the paid version, and I'm sure RStudio would be happy to advise you if you were thinking about paying for a license.

Running Shiny app on Amazon AWS

We have thoroughly discussed how to develop Shiny apps. Now it's time to discuss deploying the Shiny app on Amazon AWS. The deployment of the application is necessary to make it available for users. To deploy it on Amazon AWS, we have to follow these steps:

1. Register with Amazon AWS and opt for EC2. Amazon provides a free subscription for which the user has to provide credit card details. Such schemes are subject to change as per company policy.
2. Login to your account, go to AWS service, and then to compute and find EC2. Click on EC2 and a dashboard will appear. Click on Launch Instance and it will divert to choose **Amazon Machine Image (AMI)**. Suppose we have chosen the Ubuntu server. Now it's time to choose the instance type, which can be `t2.small`, `t2.micro`, `t2.large`, and more. Click on the launch button and another window will come for review launch. Review and click on launch.

We can also configure security by Configure Security Group. In this SSH row can be change to MYIP. Click on add rule, and add a custom TCP rule. Under port range, change it to `3838`, which is the port for Shiny server.

After this, click on review and launch and you will get a dialogue box for generating a private key. Here, we have to give a name to the key and click to download the key pair. After that, we will get the `.pem` file. Save it securely. Click on launch and get the EC2 instance running successfully. Copy the IP address available under the DNS(IPv4).

3. Access the EC2 that we created. Download putty, and convert the `.pem` file into `ppk`. Click on Puttygen, then File tab, and load the private key. Navigate to `.ppk` and import it. Save the private key with a name to your desired location. After doing all this, we have our `.ppk` file. Open putty, and in the host name box, enter the IP address of the EC2 instance. Navigate to `Auth` for

authentication and import the `ppk` key that we created. Click on `open` and enter your credentials.

4. Download `winscp`, type the `EC2` IP, and click on `advance`. Under SSH Authentication, enter the private key and click on Ok. Enter name as IP address and username and password. Click Yes in the dialogue box that appears.
5. Go to the root. Go to the prompt and type `sudo -i`, then we will get the `#` symbol. Run the following commands:

```
sudo apt-get update
sudo apt-get install r-base
sudo apt-get install r-base-dev
sudosu - -c "R -e \"install.packages('shiny', repos =
'http://cran.rstudio.com/')\""
```

To install Shiny server, run the following command:

```
wget https://download3.rstudio.org/ubuntu-12.04/x86_64/shiny-
server-1.4.4.807-amd64.deb
sudodpkg -i shiny-server-1.4.4.807-amd64.deb
```

6. We can see the folder `shiny-server` in the `/srv/shiny-server/` path. Execute the following commands:

```
sudochmod 777 /srv/shiny-server
sudomkdir /srv/shiny-server/Myapp_components
```

In the preceding command, the `Myapp_components` folder is created to save the various shiny app components, such as `server.R` and `ui.R`.

7. As we have already installed and configured `winSCP` and created the `Myapp_components` folder, we are ready to send the components to the EC2 instance. Configure the `shiny-server.conf` file, which is located in `/etc/shiny-server/`. Execute the following command:

```
| sudochmod 777 /etc/shiny-server
```

Now we can copy `configfile` onto the local system, edit it, and save it back.

8. Go to the Amazon console and find the running EC2 instance. Copy the public DNS, `ec2-34-215-115-68.us-west-2.compute.amazonaws.com`, append it with

3838/Myapp_components, and paste this in your browser's URL bar.
Press *Enter* and your app should start running.

Scoping, loading, and reusing data in Shiny applications

Although loading and reusing data in Shiny applications is covered in this chapter, because it is likely that these features would be used in a `shinyapps.io`-hosted application, much of it also applies to locally-run Shiny applications. If you use `shinyapps.io`, it will expect your application to be portable, that is, to avoid dependence on writing permanent changes to the local filesystem. This is because the application might be moved to another server for load-balancing purposes, rendering changes to the previous local filesystem inaccessible. You can write temporary files while a user is connected to the application (for example, if the user uploads their own data, this can be saved temporarily), but any changes made will be lost when the user exits.

Depending on the environment in which you are running and the task you are carrying out with your Shiny application, it is usually a good practice in most cases to make all Shiny applications portable. By making the application portable, you can not only seamlessly switch to `shinyapps.io` (even if it is just to share a beta version with colleagues using a free account), but it also means that the application is portable across other contexts; for example, if you distribute it via a `.zip` file, change your computer, or migrate the server on which you run Shiny Server. It is important, therefore, to understand the scoping of data within Shiny applications as well as the means of getting data in and out, both temporarily and permanently.

Temporary data input/output

There are three levels of scoping within the temporary, portable part of a Shiny application:

- The lowest level is data that is read-only within each individual instance of a Shiny application. This level is quite useful if a user wants a fresh copy of data each time they visit the application (if data needs to be even fresher than this, it can be placed in a reactive function). Any data loaded after the `shinyServer()` function will be scoped like this. Note that this data is only available from the `server.R` file and not from the `ui.R` file, which is loaded first.
- The next level up is data that is available to all instances of the application (again, just within `server.R`). This can be useful if there is a very large dataset that needs to be loaded or that needs significant processing; this can then be done the first time the application spins up, so users do not have to wait for it. Any data loaded before the `shinyServer()` function will be scoped in this way.
- Lastly, it is possible to make data available to `ui.R` and `server.R` across all functions by loading it in a file called `global.R`. It isn't very often that you would want to do this; I never have, but you may find it useful if you wish to configure your UI using data but don't want or need the extra code and processing time a dynamic UI would necessitate.

Remember that it is very easy to get data in and out of Shiny sessions (that is, temporarily) using the `fileInput()` and `downloadHandler()` functions.

Persistent data storage

In data science product development, one of the most important steps is to bring data from various sources and keep it on storage systems. Mostly, data-storage management for data science projects is done with a data warehouse. Nowadays, various technologies have been developed to store and process various types of data, which can be structured, semi-structured, or unstructured. Using Hadoop, HDFS, Hive, MongoDB, SQLite, or MySQL-like tools and technologies are coupled up to develop an ecosystem to make for easy availability and fast processing of data.

In normal software, the data sources are usually RDBMS and meant to deal with online transactional requirements. But in data science projects, the scenarios are quite different. Here, generally historical data is used to present graphs or generate reports. And since Shiny is also considered a tool to present data, it is less likely to expect that its apps will accept data from users all the time. But this case also needs to be considered. In this section, we will discuss the persistent data-storage options available with shiny apps and in what situations they are going to be used.

As of now, whatever Shiny apps we have developed are using inbuilt data. Such data is kept locally where the app resides. For very small data, this technique can work for some time. But as the data grows, the app will be unmanageable. In such scenarios, we need to look for other options that can handle the growing demand of data. Today's world has surpassed the big data limits. So, what are the options available?

Let's discuss some of the persistent data-storage options available:

- **Local filesystem:** This is a very easy way to store data where the app resides. If we have configured the shiny server in-house for the deployment of the app, we can take some memory to save the data accepted from user or generated during the execution. With this method, data can be saved and accessed very fast. Suppose we want to save data into a data frame and keep adding data into this table. We can follow some easy steps and

develop our app:

1. With a new Rscript file, create a dataframe:

```
| NewDF<- data.frame("Sn", "Name", "Age")
```

After the execution of the preceding code, `NewDF` will be ready to use.

Now if we want to add data to this `NewDF`, we have to write the following simple code:

```
| #insert data in first row.  
| NewDF[1,<-c(1,"Rahul",25)  
| # insert data in second row  
| NewDF[2,<-c(2, "Sumit", 26)
```

2. In the `NewDF[1,]` code, the first part of the subscript has the row number and the second column number. `NewDF[1,]` is referring to the first row and all the columns. Similarly, `NewDF[2,]` refers to the second row and all the columns. In this way, data can be added, modified, and deleted with the help of the shiny app. And at the same time, data is residing on a persistent storage. This is one of the most important methods, because whichever ecosystem is followed for data-warehousing, some part of the data has to be kept on Shiny local storage for processing.

- **Dropbox:** Dropbox is a very popular storage system available remotely. It supports a variety of file formats. `rdrop2` is a package to access Dropbox from R. It provides functions for listing files, copying, moving, and deleting. To access Dropbox, we have to first create an account. The basic subscription is free and provides 2 GB space for use. Please visit <https://www.dropbox.com/individual> for a basic subscription. To use data from Dropbox, we need to go through the authentication process first:

```
| library(rdrop2)  
| drop_auth()
```

After executing the preceding code, it will launch the browser and ask for your Dropbox account details. We have to log in to our account. Once this process is done, close the browser and complete the authentication from R. The credentials can be automatically

cached and used in future. We can also save the token using the following code:

```
| token<- drop_auth()  
| saveRDS(token, file = "token.rds")
```

We can also retrieve our account information using the `drop_acc() %>% data.frame()` command. Most of the commands to access Dropbox from R start with `drop_` like `drop_acc()`. For directory-listing, `drop_dir()` can be used. If you want to upload a file to Dropbox, we can use the following code:

```
| write.csv(newFile, "newfile.csv")  
| drop_upload("newfile.csv")
```

In a similar way, `drop_create()`, `drop_download()`, `drop_delete()`, `drop_move()`, and `drop_copy()` can also be used. All these discussed functions can be used with RShiny apps.

- **Amazon AWS S3:** A popular platform for file hosting. It keeps files in buckets. `aws.s3` is the package provided with R to access AWS S3. The `aws.s3` package is not available on CRAN. The following code can be used for installation:

```
| # latest stable version  
| install.packages("aws.s3", repos = c("cloudyr" =  
| "http://cloudyr.github.io/drat"))  
| # on windows you may need:  
| install.packages("aws.s3", repos = c("cloudyr" =  
| "http://cloudyr.github.io/drat"), INSTALL_opts = "--no-multiarch")
```

To use this package, we need to create an account with Amazon AWS for S3. A free account can also be created with limited access for one year. Once we get registered, we can generate key pairs on the IAM Management window with the heading access key. You can also visit <https://github.com/cloudyr/aws.signature/> for more detailed descriptions of the credentials. It can also be done through R using the following code:

```
| Sys.setenv("AWS_ACCESS_KEY_ID" = "mykey",  
|           "AWS_SECRET_ACCESS_KEY" = "mysecretkey",  
|           "AWS_DEFAULT_REGION" = "us-east-1",  
|           "AWS_SESSION_TOKEN" = "mytoken")
```

After setting credentials, we can access buckets using the following code:

```
| "library(\"aws.s3\")  
| bucketlist()
```

We can also access publicly-listed buckets:

```
| bucketlist()get_bucket(bucket = '1000genomes')
```

To get a list of all the objects in the private S3 bucket, the following code is useful:

```
# specify keys  
get_bucket(  
  bucket = 'my_bucket',  
  key = YOUR_AWS_ACCESS_KEY,  
  secret = YOUR_AWS_SECRET_ACCESS_KEY  
)  
# specify keys as environment variables  
Sys.setenv("AWS_ACCESS_KEY_ID" = "mykey",  
          "AWS_SECRET_ACCESS_KEY" = "mysecretkey")  
get_bucket("my_bucket")
```

The `aws.s3` package has many functions. Some of them are as follows:

- `bucketlist()`: Provides a data frame of users' buckets.
- `get_bucket()`: Provides a list and data frame, respectively, of objects in a given bucket.
- `get_bucket_df()`: Provides a list and data frame, respectively, of objects in a given bucket.
- `object_exists()`: Provides a logical output for whether an object exists.
- `s3read_using()`: Provides a generic interface for reading from S3 objects using a user-defined function.
- `get_object()`: Returns a raw vector representation of an S3 object.
- `s3connection()`: Provides a binary readable connection to stream an S3 object into R.
- `SQLite`: A lightweight database engine available publicly. It supports structured data-management. We can create tables and perform update, deletion, and insertion operations on it. It also supports transaction management.

To embed SQLite into R, the `RSQLite` package needs to be installed. It is available on CRAN:

```
| install.packages("RSQLite")
```

A connection to SQLite can be created using the following command:

```
| library(DBI)
| con<- dbConnect(RSQLite::SQLite(), ":memory:")
| dbListTables(con)
```

Using the preceding connection object, `con`, we can write the dataset in SQLite, as shown in the following snippet:

```
| dbWriteTable(con, "iris", iris)
| dbListTables(con)
```

To fetch the columns of the table:

```
| dbListFields(con, "iris")
```

And to fetch the entire table:

```
| dbReadTable(con, "iris")
```

After using of the connection, we must close it using `dbDisconnect(con)`:

- **MySQL:** MySQL is a very popular RDBMS database-management tool available publicly. It is similar to SQLite but more powerful. It can be hosted locally or remotely. We can use the `RMySQL` package to interact with MySQL and R. The package can be downloaded from CRAN:

```
| install.packages("RMySQL")
```

To install in the Linux platform or OS X, you need the MariaDB Connector. `RMySQL` acts as an interface between R and MySQL:

```
| library(RMySQL)
| library(DBI)
| con<- dbConnect(RMySQL::MySQL(), group = "my-db")
```

In the following code, `dbConnect()` is the function to establish a connection with the MySQL database from R. The `con` object can now be used to perform various operations on Database. Let's see some of them:

```
| dbListTables(con)
```

The preceding code lists the tables in the database. To write a table from R into MySQL, use the following code:

```
| dbWriteTable(con, "iris", iris)
| dbListTables(con)
```

We can also fetch the entire table from the database as follows:

```
| # You can fetch all results:  
| res<- dbSendQuery(con, "SELECT * FROM iris ")  
| dbFetch(res)
```

To disconnect the connection, use the following code.

```
| dbDisconnect(con)
```

- **Google Sheets:** Google sheets is also a good option for storing data. It is persistent and accessible from anywhere. Google sheets can be accessed using a title, key, or URL. Data can be easily extracted or edited. We can create, delete, rename, copy, and upload Google sheets. We can also upload locally-generated spreadsheets into Google sheets and vice versa.

To use Google sheets from R, we can use the `googlesheets` package. It is developed to be used with the `%>%` pipe operator. It uses the `dplyr` package internally. It can be downloaded from CRAN:

```
| install.packages("googlesheets")
```

With this package, a demo sheet for practice is available, named `Gapminder`. Let's use some code to play with this sheet:

```
| gs_gap() %>%  
|   gs_copy(to = "Gapminder")
```

This code is for the copying of the sheet. To view the sheet in the browser:

```
| gap %>% gs_browse()  
| gap %>% gs_browse(ws = "Europe")
```

To read all the data in the worksheet, we can use the following code:

```
| africa<- gs_read(gap)  
| glimpse(africa)  
| Africa
```

We can also target specific cells:

```
| gap %>% gs_read(ws = 2, range = "A1:D8")
```

We can also create new spreadsheets:

```
|giris_ss<- gs_new("iris", input = head(iris, 3), trim = TRUE)
```

We can explore more about google sheets and R at <https://github.com/jennybc/googlesheets#install-googlesheets>.

Database using Dplyr, DBI, and POOL

In this section, we will learn to use the `dplyr` package to access data from database sources. We will also see how to hook up to an external database using the `DBI` package. The Pool package is also an important topic to manage connections and prevent leaks to manage performance.

- `dplyr`: A popular data-manipulation package for internal and external databases. It internally works as SQL. It provides a variety of functions for data manipulation:

- `filter()`
- `select()`
- `arrange()`
- `rename()`
- `distinct()`
- `mutate()`
- `transmute()`
- `summarise()`
- `sample_n()`
- `sample_frac()`

Let's see an example using some of these functions with the `iris` dataset:

```
| library(dplyr)
| iris %>% filter(Sepal.Length>4 & Sepal.Length<5)
```

In the preceding code, the `filter` function has been used to filter the rows of the `iris` dataset, which has values between 4 and 5. We can also select only certain columns, using `select()`:

```
| iris %>% select(Sepal.Length)
```

For selecting distinct values, we can use `distinct()`:

```
| iris %>% distinct()
```

We can also use these functions in combination. The code shown here is first filtering the data and then applying `select()`:

```
| iris %>% filter(Sepal.Length>4 & Sepal.Length<5) %>% select(Sepal.Length)
```

In this way, we can get ride of using R and SQL separately using `dplyr`.

- DBI: A common interface between R and DBMS. It also provides a variety of functions for supporting the following functions:
 - Connecting and disconnecting to the DBMS
 - Creating statements and executing in the DBMS
 - Extracting results from statements
 - Exception-handling
 - Transaction management

`dbGetQuery()` is a convenient way of executing queries with a connection object. It takes two arguments importantly connection to database and query:

```
| dbGetQuery(conn, "SELECT * FROM City LIMIT 5;")
```

- Pool: With the DBI package, there are some problems with connection-management and performance. We have to create and destroy connections as and when necessary. Otherwise, there will be an accumulation of leaked connections. This leaked connection holds resources that must have been freed. Because of this app, the performance goes down. For dealing with such problems, `pool` package is available. It provides an extra layer of abstraction while establishing a connection. Using this package, we can create an object with a reference to the database. This object is called `pool`. This makes us avoid directly fetching the database. Also, this `pool` object can hold a number of connections to the database. Whenever we are querying the database, we are actually querying to the pool that holds the running and waiting connections.

The `pool` can provide us with idle connections that were previously fetched or with a new one. Once the connection is ended, the garbage-collection process makes free all resources held by the connection. So, in the context of the Shiny app, we don't have to worry about ending the connection.

The `pool` package is available on GitHub and can be downloaded as follows:

```
| devtools::install_github("rstudio/pool")
```

Let's see the skeleton of a Shiny app with `pool`:

```
library(shiny)
library(DBI)
library(pool)
pool<- dbPool(
  drv = RMySQL::MySQL(),
  dbname = "Database_Name",
  host = "Host_link",
  username = "username",
  password = "password"
)
ui<- fluidPage(
  #----- Ui stuff-----
)
server<- function(input, output, session) {
#---Server Stuff-----
}
shinyApp(ui, server)
```

With `sqlInterpolate()`, we can create a query that can be the input to `dbGetQuery()`. In `sqlInterpolate`, we can see three parameters in the function: `pool`, `sql`, and `id`. `pool` is the object and `sql` is the SQL query. `id` can be any input. `dbGetQuery()` will have two parameters. The `pool` object and query are created using `interpolate`:

```
| query<- sqlInterpolate(pool, sql, id = input$ID)
| dbGetQuery(pool, query)
```

This app skeleton is for a single-file app. For a multi-file app, it can be put on the top of the `server` and `ui` file or in a global file.

SQL Injection

SQL Injection is a kind of attack done by adding SQL queries to the URL of the application. Such queries execute on the DBMS without having legitimate access to it. Such attacks are possible if there are some branches into the code. Let's see some code to understand it better:

```
| dbGetQuery(conn, paste0( "SELECT * FROM City LIMIT ", input$nrows, ";"))
```

As we can see in the preceding code, `input$nrows` has been put directly into the query. If an attacker got access to this `input$nrows`, they could inject any SQL statement into it. In this case, the solution can be to prevent an attacker from passing vectors. So, the code can be modified as follows:

```
| dbGetQuery(conn, paste0( "SELECT * FROM City LIMIT ", as.integer(input$nrows)[1], ";"))
```

The input is converted into an integer first. So, if an attacker puts some SQL into it, it will get converted into an integer and lose its meaning. This is an easy example. Let's discuss a more complex situation:

```
| query<- paste0("SELECT * FROM City WHERE ID = '", input$ID, "';")
```

In this code, if an attacker gets access to `input$ID` and manages to modify the value, they can get the data of a single city. For example, `input$ID` changed to `5` means that the data of the city with `ID=5` will be invoked. But if they are trying to get information of all the cities, they can try `OR 1 = 1 OR` and can get data of all the city. Such input makes the condition always true. The solution to this attack is to use `sqlInterpolate()`. It can be used to interpolate the values into a SQL string, which prevents a SQL injection attack:

```
| sql<- "SELECT * FROM City WHERE ID = ?id ;"  
| query<- sqlInterpolate(conn, sql, id = input$ID)
```

In the preceding code, if we change the input to '`OR 1 = 1 OR`', it will be converted into `SELECT * FROM City WHERE ID = '' OR 1 = 1 OR ''`, where it has added extra `('')`. This will give blank table output and prevents SQL injection. Instead, if we use `input$ID = 6`, it will change the query to `SELECT * FROM City WHERE ID = '6'`, which is a valid ID. So the output will be the data of the city with `ID=6`.

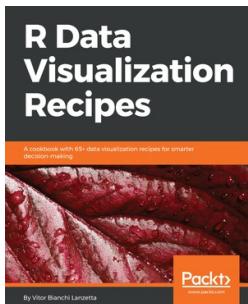
This process of checking and converting user input into safe values is called **sanitization**. We must always take care to sanitize the user input to prevent SQL injection attacks.

Summary

In this chapter, we learned about several methods to share your Shiny applications with the world. This process is very easy with fellow users of R, and a little harder with the whole internet; however you do it, I'm sure you'll agree that it was relatively painless and worth the effort. We discussed how to use Git and GitHub (and Gist), and how to use them to share your code and applications with other R users. We also looked at distributing Shiny applications manually or over FTP to R users using the `.zip` and `.tar` files. We covered hosting solutions to share your application with the whole internet, including Shiny apps, Shiny Server, and Amazon AWS. We went through the persistent data-storage options and how to use the `dplyr`, `DBI`, and `pool` packages. We also saw how to prevent SQL injection attacks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

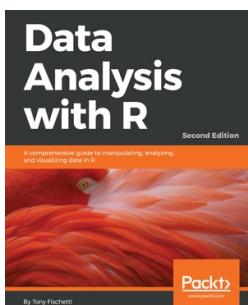


R Data Visualization Recipes

Vitor Bianchi Lanzetta

ISBN: 9781788398312

- Get to know various data visualization libraries available in R to represent data
- Generate elegant codes to craft graphics using ggplot2, ggviz and plotly
- Add elements, text, animation, and colors to your plot to make sense of data
- Deepen your knowledge by adding bar-charts, scatterplots, and time series plots using ggplot2
- Build interactive dashboards using Shiny.
- Color specific map regions based on the values of a variable in your data frame
- Create high-quality journal-publishable scatterplots
- Create and design various three-dimensional and multivariate plots



Data Analysis with R - Second Edition

Tony Fischetti

ISBN: 9781788393720

- Gain a thorough understanding of statistical reasoning and sampling theory
- Employ hypothesis testing to draw inferences from your data
- Learn Bayesian methods for estimating parameters
- Train regression, classification, and time series models
- Handle missing data gracefully using multiple imputation
- Identify and manage problematic data points
- Learn how to scale your analyses to larger data with Rcpp, data.table, dplyr, and parallelization
- Put best practices into effect to make your job easier and facilitate reproducibility

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!