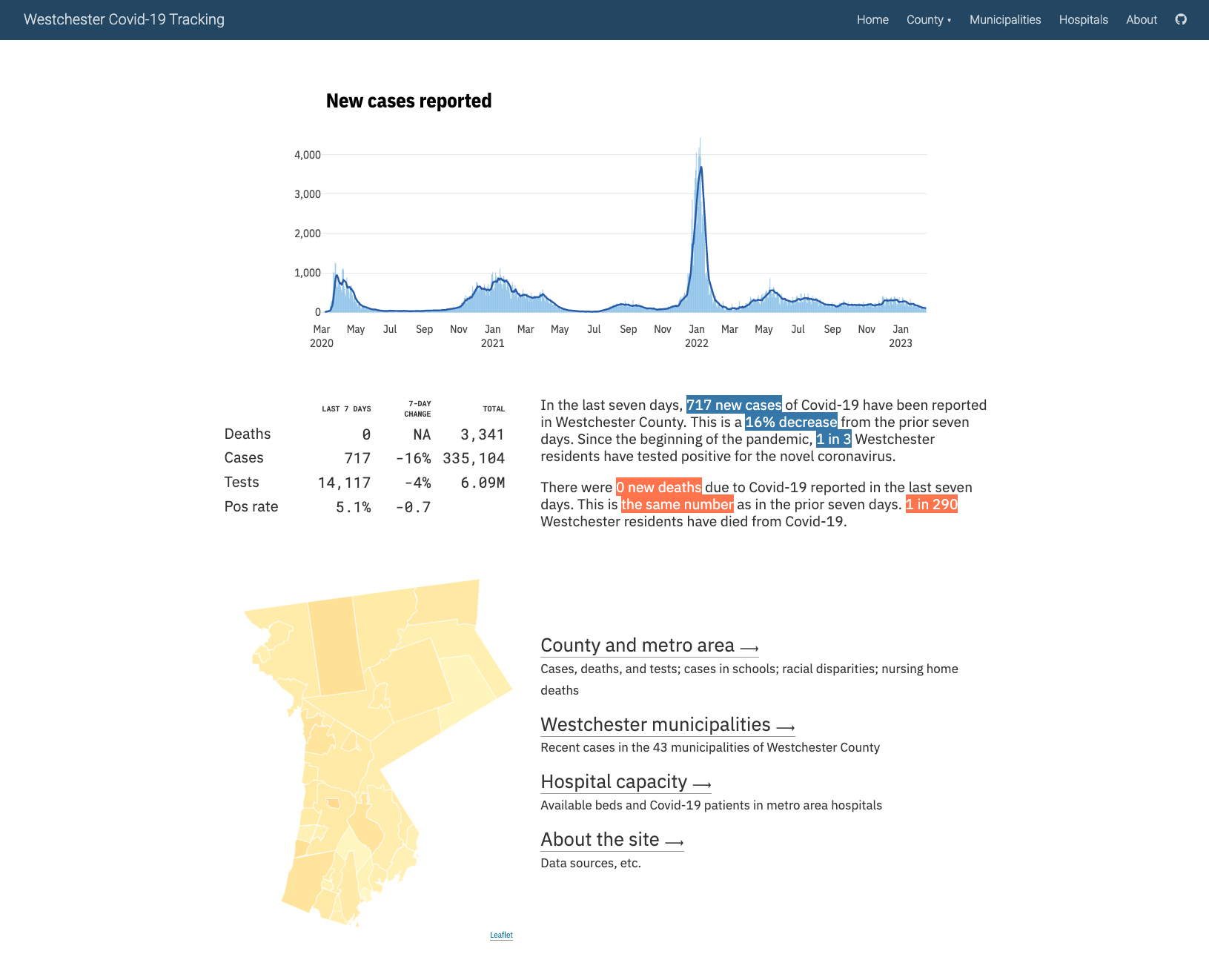
Building Websites with distill

In the summer of 2020, Matt Herman’s family moved from Brooklyn to Westchester County, New York. It was early in the COVID pandemic, and Herman was shocked that the county published little data about infection rates. Vaccines weren’t yet available, and daily choices like whether to take children to the park depended on access to good data.

Matt Herman wasn’t just any person with an interest in COVID data. He was also, at the time, Deputy Director of the ChildStat Data Unit in the Office of Research and Analytics at the New York City Administration for Children’s Services. This handful of a title meant he was skilled at working with data, enabling him to create the COVID resource he needed: the Westchester COVID-19 Tracking website.

Built entirely in R, this website uses charts, maps, tables, and text to summarize the latest COVID data for Westchester County. The website, available at <https://westchester-covid.mattherman.info/>, is no longer consistently updated, but Figure 9-1 shows a screenshot of it.

[F09001.png]



* + - * 1. A screenshot of the Westchester COVID-19 website

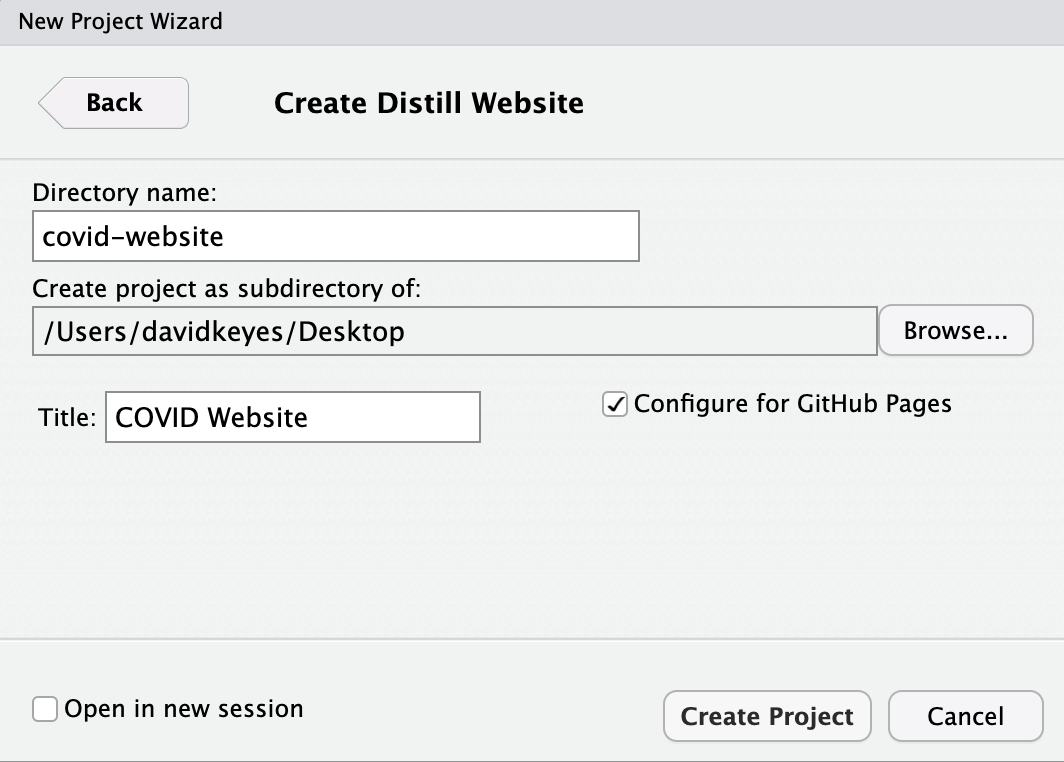
To make this website, Herman wrote a set of R Markdown files and strung them together with the distill package. This chapter explains the basics of the package by walking through the creation of a simple website. You’ll learn how to produce different page layouts, navigation menus, and interactive graphics, then explore strategies for hosting your website.

Creating a New distill Project

In Chapter 8, you created a slideshow presentation by knitting an R Markdown document to a single HTML file. A website is merely a collection of HTML files. The distill package enables us to use multiple R Markdown documents to create several HTML files, then connect them with a navigation menu and more.

To create a distill website, install the package using install.packages("distill"). Then start a project in R Studio by navigating to File⏵New Project⏵New Directory and selecting Distill Website as the project type. This should take you to the menu in Figure 9-2.

[F09002.png]



* + - * 1. Giving your distill website a name

Specify the directory and subdirectory where your project will live on your computer, then give your website a title. The Configure for GitHub Pages option provides an easy way to get your website online, and we’ll discuss how it works in “Using GitHub Pages” on page XX. Select it if you’d like to use this deployment option.

The Project Files

You should now have a project with several files. In addition to the *covid-website.Rproj* file that indicates that we’re working in an RStudio project, we have two R Markdown documents, a *\_site.yml* file, and a *docs* folder, where the rendered HTML files will go. Let’s take a look at these website’s files.

The R Markdown Documents

Each R Markdown file represents a page of the website. By default, distill creates a Home page (*index.Rmd*) and an About page (*about.Rmd*) containing placeholder content. If you wanted to generate additional pages, you would simply add new R Markdown files, then add them to the *\_site.yml* file (discussed in the next section).

If you open the *index.Rmd* file, you’ll notice that the YAML contains two parameters not present in the R Markdown documents from previous chapters: description and site:

---

title: "COVID Website"

description: |

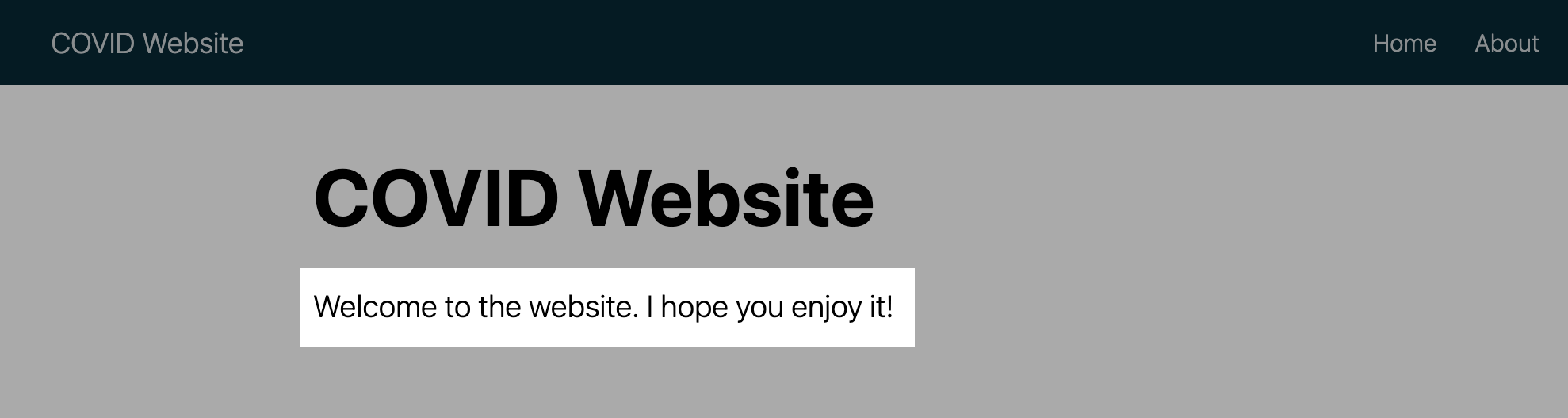
Welcome to the website. I hope you enjoy it!

site: distill::distill\_website

---

The description parameter specifies the text that should go below the title of each page, as shown in Figure 9-3.

[F09003.png]



* + - * 1. The default website description

The site: distill::distill\_website line identifies the root page of a distill website. This means that when we knit the document, R Markdown knows to create a website, rather than an individual HTML file, and that the website should display this page first. Other pages of the website don’t require this line. As long as they are listed in the *\_site.yml* file, they will be added to the site.

You’ll notice we’re also missing a parameter we’ve seen in other R Markdown documents: output, which specifies the output format to use while knitting. The reason we don’t have this parameter here is that we specify the output for the entire website in the *\_site.yml* file.

The \_site.yml File

The *\_site.yml* file tells R which R Markdown documents make up the website, what the knitted files should look like, what the website should be called, and more. If you open it, you should see the following code.

name: "covid-website"

title: "COVID Website"

description: |

COVID Website

output\_dir: "docs"

navbar:

right:

- text: "Home"

href: index.html

- text: "About"

href: about.html

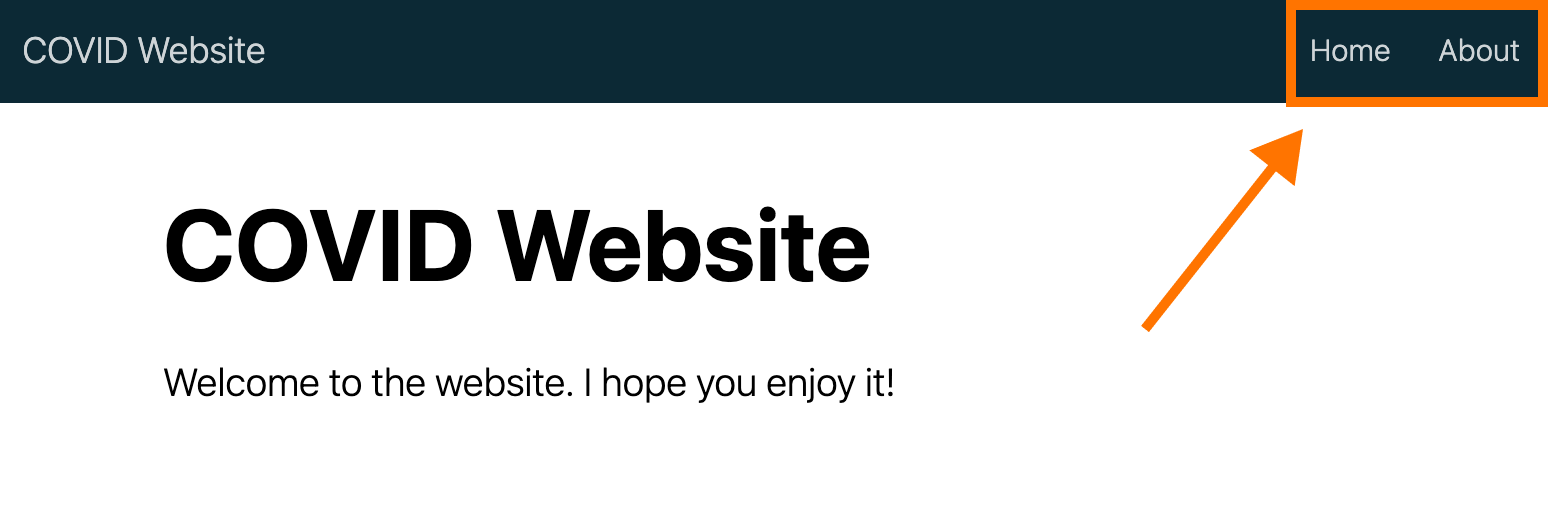
output: distill::distill\_article

The name parameter determines the URL for your website. By default, this should be the name of the directory in which your distill project lives. The title parameter creates the title for the entire website and shows up in the top left of the navigation bar by default. The description parameter provides what’s known as a *meta description*, which will show up on Google search results as a couple of lines to give the user an overview of the website content.

The output\_dir parameter determines where the rendered HTML files live when we generate the website. You should see the .*docs* directory listed here if you selected the Configure for GitHub Pages option. However, you can change the output directory to any folder you choose.

Next, the navbar section defines the website’s navigation. Ours appears on the right side of the header, but swapping the right parameter for left would switch its position. It includes links to the site’s two pages: Home and About, as shown in Figure 9-4.

[F09004.png]



* + - * 1. The website navigation bar

Within the navbar code, the text parameter determines what text shows up in the menu. (Try, for example, changing “About” to “About this Website.”) The href parameter determines which HTML file the text in the navigation bar links to. If you want to include additional pages on your menu, you’ll need to add both the text and href parameters.

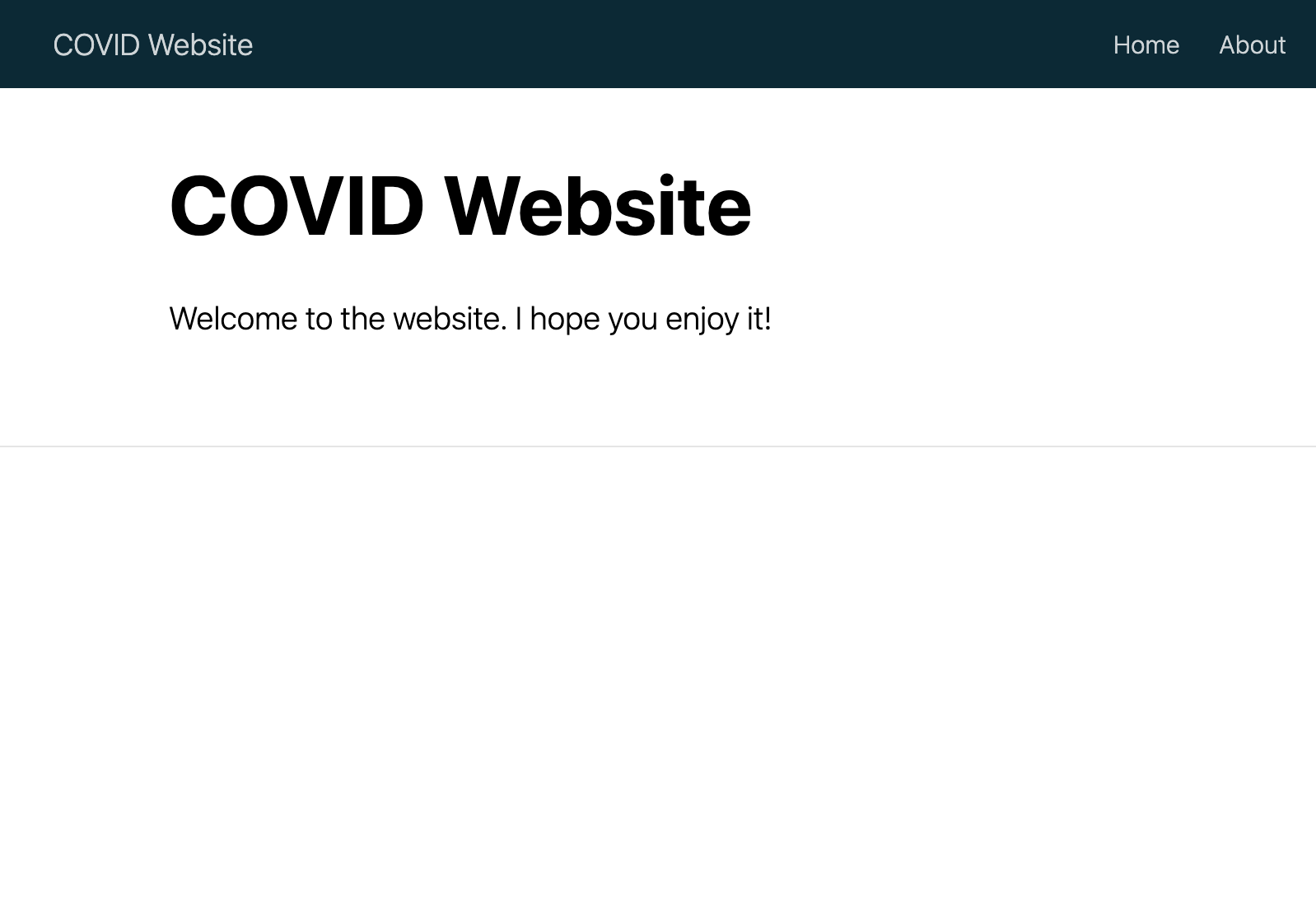
Lastly, the output parameter specifies that all R Markdown documents should be rendered using the distill\_article format. This format allows for layouts of different widths, *asides* (parenthetical items that live in a sidebar next to the main content), easily customizable CSS, and more.

Building the Site

We explored the project’s files but haven’t yet used these to create the website. There are three ways to do this. First, you can click the Build Website button in the Build tab of RStudio’s top-right pane. Second, you can run rmarkdown::render\_site() in the console or in an R script file. Third, you can use the keyboard shortcut Command + Shift + B on macOS and Ctrl + Shift + B on Windows.

These options will render all R Markdown documents and add the top navigation bar to them with the options specified in the *\_site.yml* file. To find the rendered files, look in the output directory you specified (for us, that was *docs*). Open the *index.html* file and you’ll find your website, which should look like Figure 9-5.

[F09005.png]



* + - * 1. The COVID website with default content

You can open any other HTML file as well to see its rendered version.

Applying Custom CSS with create\_theme()

Websites made with distill tend to look similar, but you can change this design using custom CSS. The distill package even provides a function to simplify the process. Running this distill::create\_theme() function in the console should create a file called *theme.css*, shown here:

/\* base variables \*/

/\* Edit the CSS properties in this file to create a custom

Distill theme. Only edit values in the right column

for each row; values shown are the CSS defaults.

To return any property to the default,

you may set its value to: unset

All rows must end with a semi-colon. \*/

/\* Optional: embed custom fonts here with `@import` \*/

/\* This must remain at the top of this file. \*/

html {

/\*-- Main font sizes --\*/

--title-size: 50px;

--body-size: 1.06rem;

--code-size: 14px;

--aside-size: 12px;

--fig-cap-size: 13px;

/\*-- Main font colors --\*/

--title-color: #000000;

--header-color: rgba(0, 0, 0, 0.8);

--body-color: rgba(0, 0, 0, 0.8);

--aside-color: rgba(0, 0, 0, 0.6);

--fig-cap-color: rgba(0, 0, 0, 0.6);

/\*-- Specify custom fonts ~~~ must be imported above --\*/

--heading-font: sans-serif;

--mono-font: monospace;

--body-font: sans-serif;

--navbar-font: sans-serif; /\* websites + blogs only \*/

}

/\*-- ARTICLE METADATA --\*/

d-byline {

--heading-size: 0.6rem;

--heading-color: rgba(0, 0, 0, 0.5);

--body-size: 0.8rem;

--body-color: rgba(0, 0, 0, 0.8);

}

/\*-- ARTICLE TABLE OF CONTENTS --\*/

.d-contents {

--heading-size: 18px;

--contents-size: 13px;

}

/\*-- ARTICLE APPENDIX --\*/

d-appendix {

--heading-size: 15px;

--heading-color: rgba(0, 0, 0, 0.65);

--text-size: 0.8em;

--text-color: rgba(0, 0, 0, 0.5);

}

/\*-- WEBSITE HEADER + FOOTER --\*/

/\* These properties only apply to Distill sites and blogs \*/

.distill-site-header {

--title-size: 18px;

--text-color: rgba(255, 255, 255, 0.8);

--text-size: 15px;

--hover-color: white;

--bkgd-color: #0F2E3D;

}

.distill-site-footer {

--text-color: rgba(255, 255, 255, 0.8);

--text-size: 15px;

--hover-color: white;

--bkgd-color: #0F2E3D;

}

/\*-- Additional custom styles --\*/

/\* Add any additional CSS rules below \*/

Within that file is a set of what are known as *CSS variables*. Most of them have names that clearly show their purpose, and you can alter their default values to whatever you’d like. For example, the following changes to the site’s header make the title and text size larger and the background color a light blue:

.distill-site-header {

--title-size: 28px;

--text-color: rgba(255, 255, 255, 0.8);

--text-size: 20px;

--hover-color: white;

--bkgd-color: #6cabdd;

}

Before we can see these changes, we need to tell distill to use this custom CSS when rendering by adding a line to the *\_site.yml* file:

name: "covid-website"

title: "COVID Website"

description: |

COVID Website

theme: theme.css

output\_dir: "docs"

navbar:

*--snip--*

Now we can generate the site again. As you can see in the *theme.css* file, there are a lot of CSS variables we can change to tweak the appearance of the website. Playing around with them and rebuilding your site is a great way to figure out what does what.

Working with Website Content

Using distill, we can alter the size of the website’s content or even make it interactive. To first add content to a page on the website, create Markdown text and code chunks in the page’s R Markdown document. For example, let’s replace the contents of *index.Rmd* with the following code to make a table, a map, and a chart appear on the website’s Home page. Here is the start of the file:

---

title: "COVID Website"

description: "Information about COVID rates in the United States over time"

site: distill::distill\_website

---

```{r setup, include=FALSE}

knitr::opts\_chunk$set(echo = FALSE,

warning = FALSE,

message = FALSE)

```

```{r}

# Load packages

library(tidyverse)

library(janitor)

library(tigris)

library(gt)

library(lubridate)

```

After the YAML and setup code chunk, we load several packages, most of which you’ve seen in previous chapters: tidyverse for data import, manipulation, and plotting (with ggplot); janitor for its clean\_names() function, which makes our variable names easier to work with; tigris to import geospatial data about states; gt for making nice tables; and lubridate to work with dates.

Next, we create a new code chunk to import and clean our data:

```{r}

# Import data

us\_states <- states(cb = TRUE,

resolution = "20m",

progress\_bar = FALSE) %>%

shift\_geometry() %>%

clean\_names() %>%

select(geoid, name) %>%

rename(state = name) %>%

filter(state %in% state.name)

covid\_data <- read\_csv("https://raw.githubusercontent.com/nytimes/covid-19-data/master/rolling-averages/us-states.csv") %>%

filter(state %in% state.name) %>%

mutate(geoid = str\_remove(geoid, "USA-"))

1 most\_recent\_day <- covid\_data %>%

slice\_max(order\_by = date,

n = 1) %>%

distinct(date) %>%

mutate(date\_nice\_format = str\_glue("{month(date, label = TRUE, abbr = FALSE)} {day(date)}, {year(date)}")) %>%

pull(date\_nice\_format)

```

2# COVID Death Rates as of `r most\_recent\_day`

This table shows COVID death rates per 100,000 people in four states states.

We also create a variable called most\_recent\_day 1, which we later reference in a text section. Using inline R code, this header now displays the current date 2. We then make a table that shows the death rates per 100,000 people in four states (as using all states would create too large a table):

```{r}

covid\_data %>%

filter(state %in% c(“Alabama”,

“Alaska”,

“Arizona”,

“Arkansas”)) %>%

slice\_max(order\_by = date,

n = 1) %>%

select(state, deaths\_avg\_per\_100k) %>%

arrange(state) %>%

set\_names(“State”, “Death rate”) %>%

gt() %>%

tab\_style(

style = cell\_text(weight = “bold”),

locations = cells\_column\_labels()

)

```

This table resembles the code we discussed in Chapter 5. Next, we make a map of this data for all states using techniques covered in Chapter 4:

We can see this same death rate data for all states on a map.

```{r}

most\_recent <- us\_states %>%

left\_join(covid\_data, by = “state”) %>%

slice\_max(order\_by = date,

n = 1)

most\_recent %>%

ggplot(aes(fill = deaths\_avg\_per\_100k)) +

geom\_sf() +

scale\_fill\_viridis\_c(option = “rocket”) +

labs(fill = “Deaths per\n100,000 people”) +

theme\_void()

```

Finally, we make a chart that shows COVID death rates over time in the four states from the table:

# COVID Death Rates Over Time

The following chart shows COVID death rates from the start of COVID in early 2020 until `r most\_recent\_day`.

```{r}

covid\_data %>%

filter(state %in% c("Alabama",

"Alaska",

"Arizona",

"Arkansas")) %>%

ggplot(aes(x = date,

y = deaths\_avg\_per\_100k,

group = state,

fill = deaths\_avg\_per\_100k)) +

geom\_col() +

scale\_fill\_viridis\_c(option = "rocket") +

theme\_minimal() +

labs(title = "Deaths per 100,000 people over time") +

theme(legend.position = "none",

plot.title.position = "plot",

plot.title = element\_text(face = "bold"),

panel.grid.minor = element\_blank(),

axis.title = element\_blank()) +

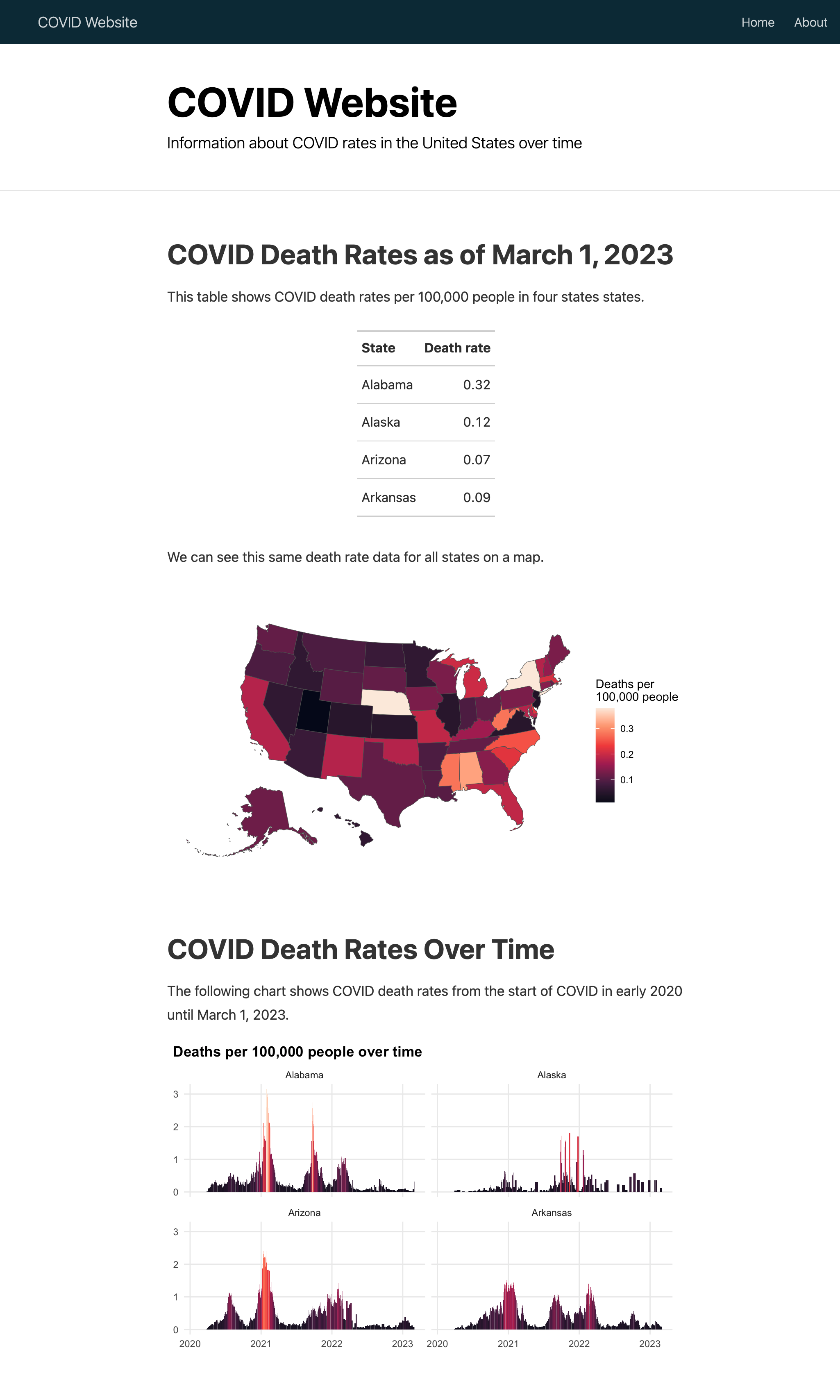
facet\_wrap(~state,

nrow = 2)

```

Figure 8-6 shows what the website’s home page now looks like.

[F09006.png]



* + - * 1. The COVID website with a table, map, and chart

Now that we have some content, we can tweak it. For example, because many states are quite small, especially in the northeast, it’s a bit challenging to see them. Let’s make the entire map bigger.

Applying distill Layouts

One nice feature of distill is the ability to make the output of certain code chunks larger. It has four layouts that you can apply to a code chunk to widen its output: l-body-outset (creates output that is a bit wider than default), l-page (creates output that is wider still), l-screen (creates full-screen output), and l-screen-inset (creates full-screen output with a bit of a buffer).

Apply l-screen-inset to the map by modifying the first line of its code chunk as follows:

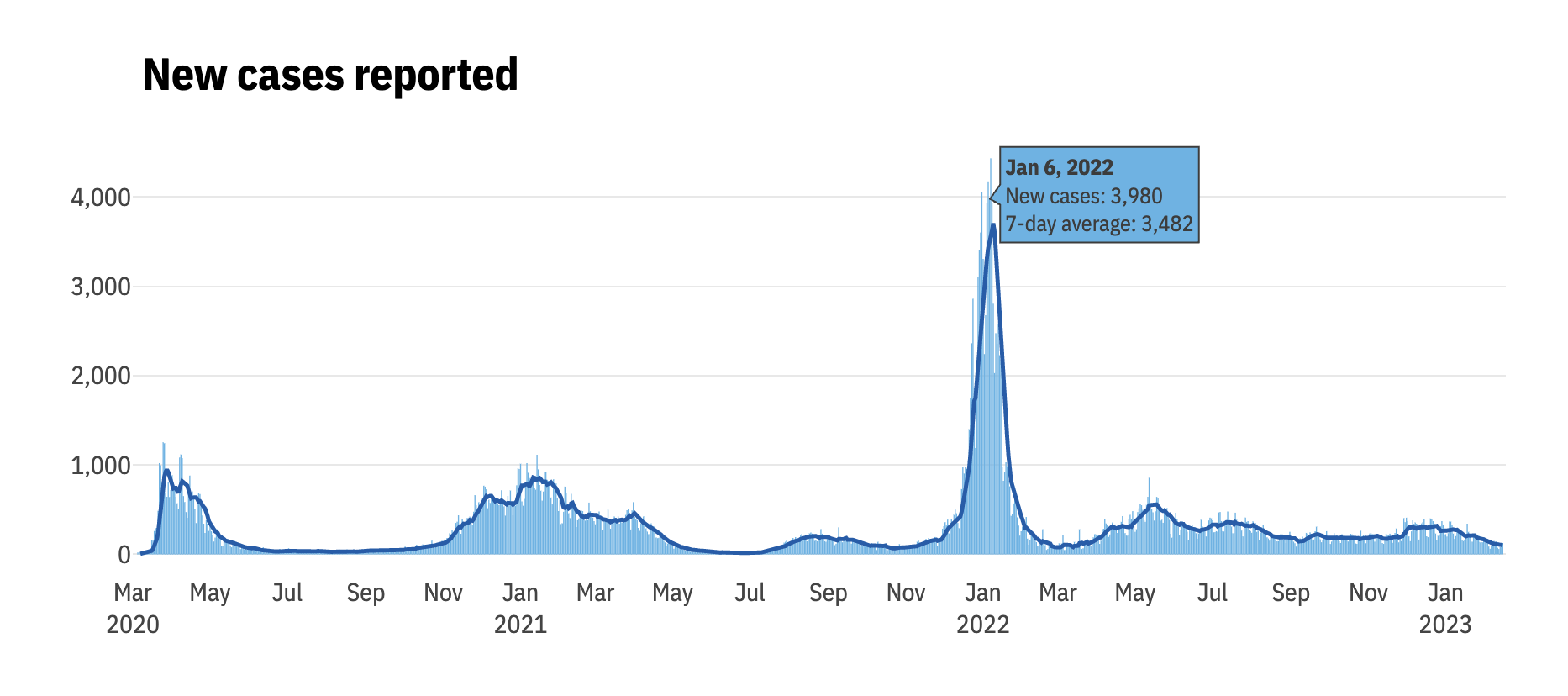
```{r layout = "l-screen-inset"}

Doing this makes our map wider and taller, and as a result, much easier to read.

Making the Content Interactive

The static content we’ve added to the website so far has none of the interactivity common in websites, which can use JavaScript to respond to user behavior. If you have limited familiarity with HTML and JavaScript, you can use R packages like distill, plotly, and DT, which wrap JavaScript libraries, to add interactivity to your website. Matt Herman uses interactive graphics and maps on his Westchester County COVID website. Figure 9-7, for example, shows a tooltip that allows the user to see results for any single day.

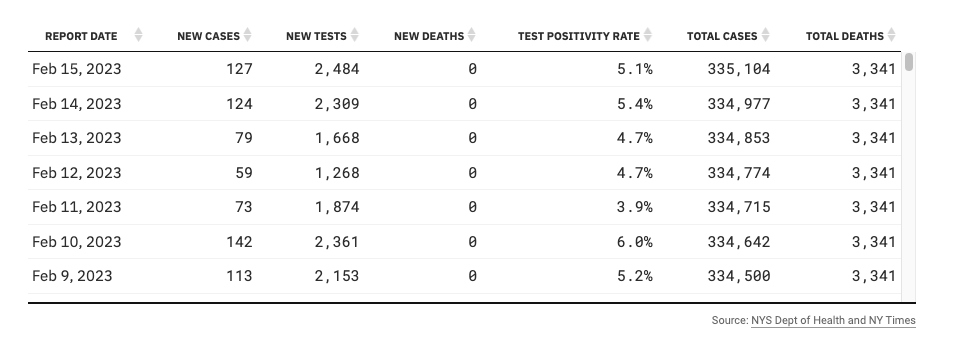
[F09007.png]



* + - * 1. Interactive tooltips showing new cases by day

Herman also makes interactive tables with the DT package, allowing the user to scroll through the data and sort the values by clicking any of the variables in the heading. Figure 9-8 shows the table.

[F09008.png]



* + - * 1. An interactive table made with the DT package

Let’s add some interactivity to our national COVID website. We’ll begin by making our table interactive.

Adding Pagination to a Table with reactable

Remember how we included only four states in the table in to keep it from getting super long? By creating an interactive table, we can avoid this. The reactable package is a great option for interactive tables. You can install it with install.packages("reactable"). If we swap out the gt package code we used to make our static table with the reactable() function, we can show all states:

library(reactable)

covid\_data %>%

slice\_max(order\_by = date,

n = 1) %>%

select(state, deaths\_avg\_per\_100k) %>%

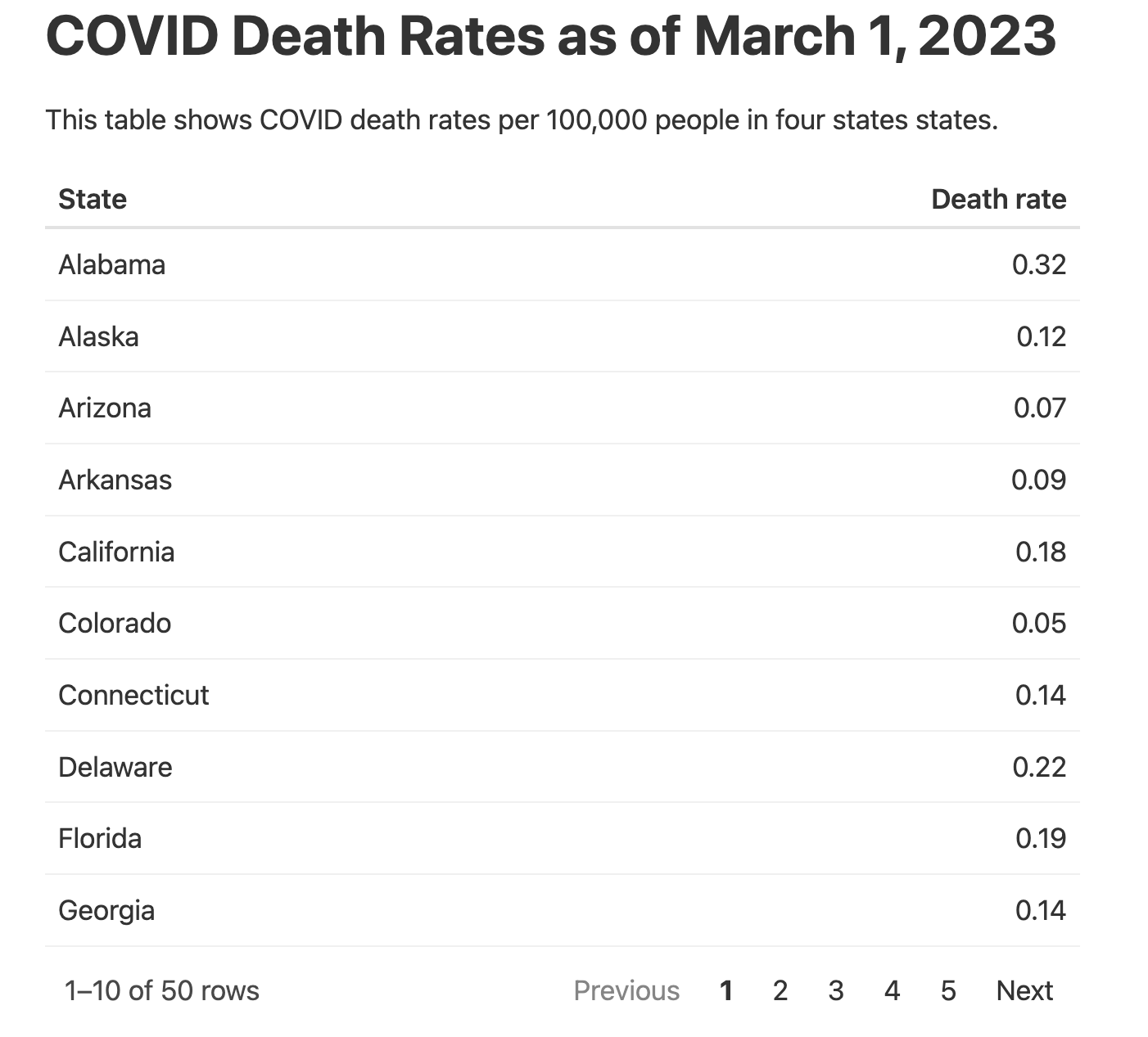
arrange(state) %>%

set\_names("State", "Death rate") %>%

reactable()

The reactable package shows 10 rows by default and adds pagination, as shown in Figure 9-9.

[F09009.png]



* + - * 1. An interactive table built with reactable

The reactable() function also enables sorting by default. Although we used the arrange() function in our code to sort the data by state name, users can click the “Death rate” column to sort values using that variable.

Creating a Hovering Tooltip with Plotly

Let’s also give the website’s chart some interactivity using plotly, which has the following basic workflow. First, create a plot with ggplot and save it as an object. Then, pass it to the ggplotly() function, which turns it into an interactive plot. In the following code, which assumes you have installed the plotly package using install.packages("plotly"), we apply it to the chart showing COVID death rates over time:

library(plotly)

covid\_chart <- covid\_data %>%

filter(state %in% c("Alabama",

"Alaska",

"Arizona",

"Arkansas")) %>%

ggplot(aes(x = date,

y = deaths\_avg\_per\_100k,

group = state,

fill = deaths\_avg\_per\_100k)) +

geom\_col() +

scale\_fill\_viridis\_c(option = "rocket") +

theme\_minimal() +

labs(title = "Deaths per 100,000 people over time") +

theme(legend.position = "none",

plot.title.position = "plot",

plot.title = element\_text(face = "bold"),

panel.grid.minor = element\_blank(),

axis.title = element\_blank()) +

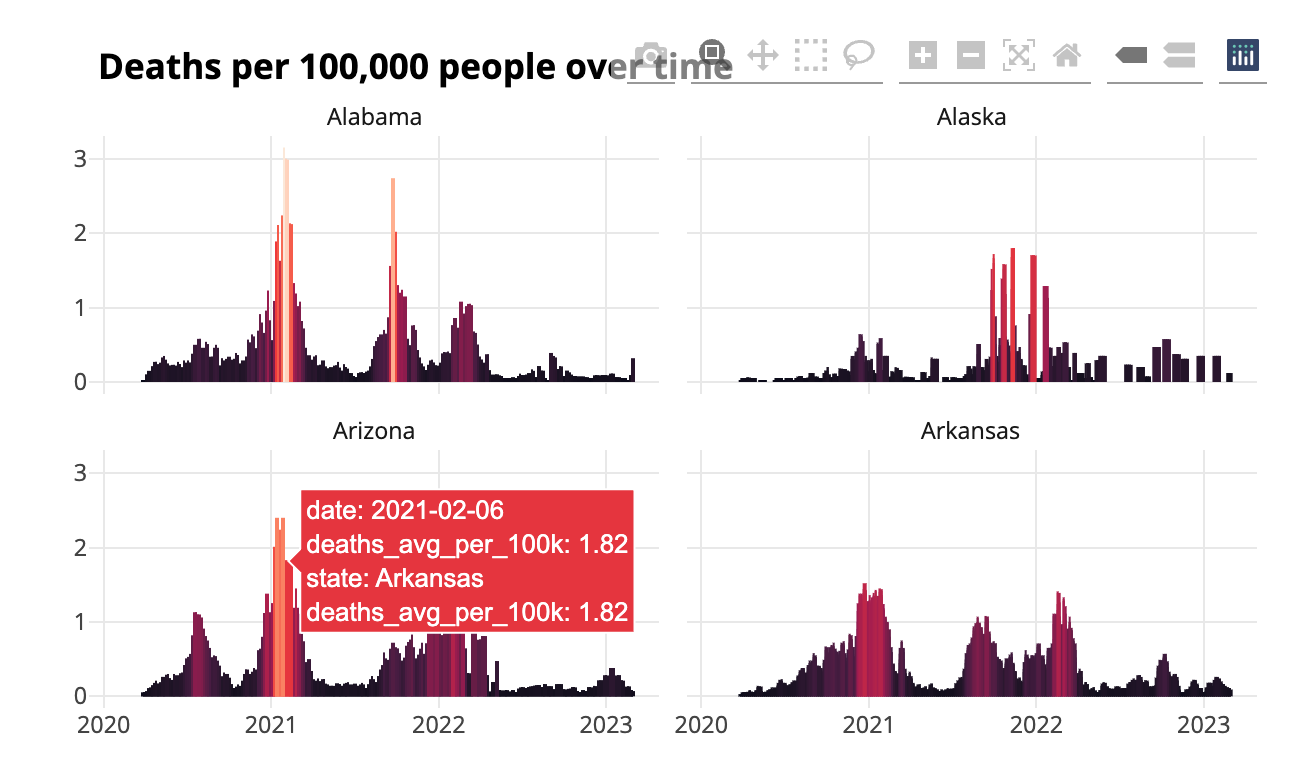
facet\_wrap(~state,

nrow = 2)

ggplotly(covid\_chart)

The code to make the chart is identical to the chart code shown earlier in this chapter. The only difference is that we save our chart as an object called covid\_chart and then run ggplotly(covid\_chart). This code produces an interactive chart that shows the data for a particular day when a user hovers over it. But the tooltip that users see when hovering, shown in Figure 9-10, is cluttered and overwhelming because the ggplotly() function shows all data by default.

[F09010.png]



* + - * 1. Messy tooltips on the COVID death rates graph

To make the tooltip more informative, we can create a single variable containing the data we want to display and then tell ggplotly() to use this:

covid\_chart <- covid\_data %>%

filter(state %in% c("Alabama",

"Alaska",

"Arizona",

"Arkansas")) %>%

mutate(date\_nice\_format = str\_glue("{month(date, label = TRUE, abbr = FALSE)} {day(date)}, {year(date)}")) %>% 1

mutate(tooltip\_text = str\_glue("{state}<br>{date\_nice\_format}<br>{deaths\_avg\_per\_100k} per 100,000 people")) %>% 2

ggplot(aes(x = date,

y = deaths\_avg\_per\_100k,

group = state,

text = tooltip\_text, 3

fill = deaths\_avg\_per\_100k)) +

geom\_col() +

scale\_fill\_viridis\_c(option = "rocket") +

theme\_minimal() +

labs(title = "Deaths per 100,000 people over time") +

theme(legend.position = "none",

plot.title.position = "plot",

plot.title = element\_text(face = "bold"),

panel.grid.minor = element\_blank(),

axis.title = element\_blank()) +

facet\_wrap(~state,

nrow = 2)

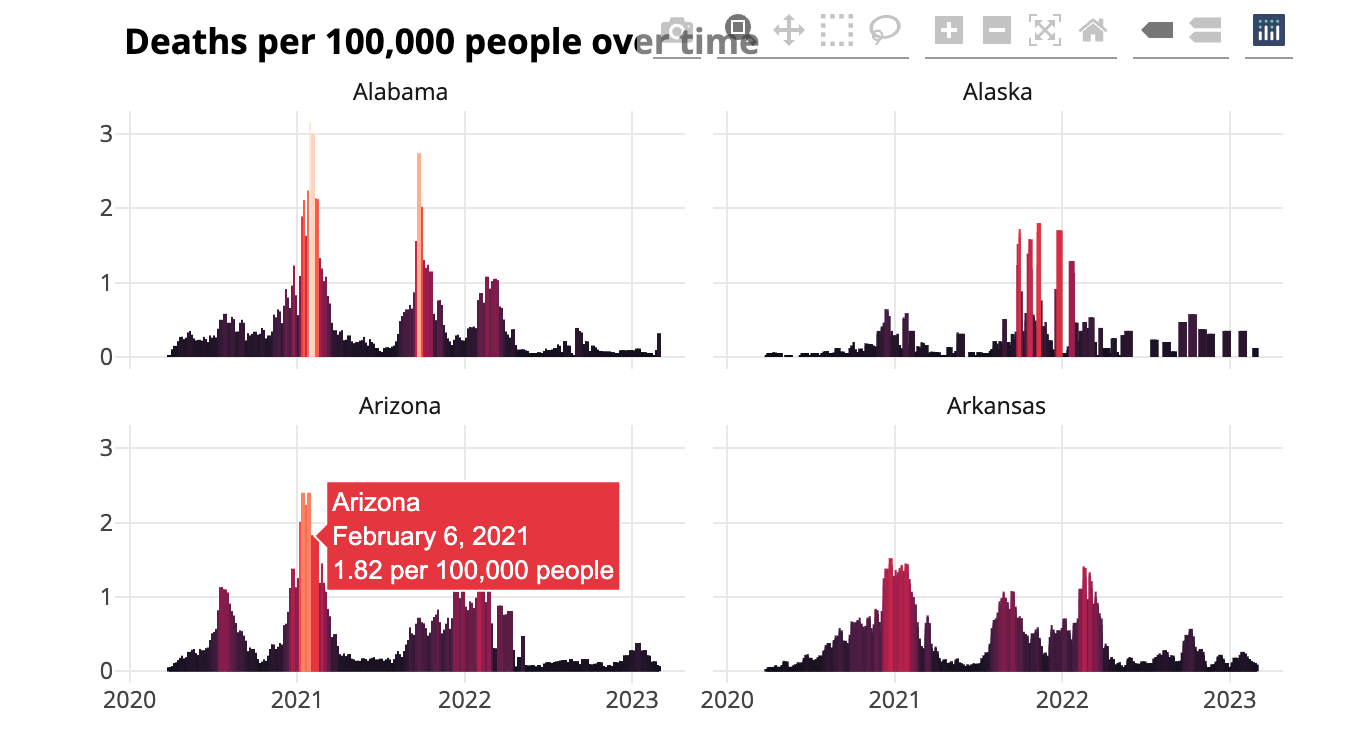
ggplotly(covid\_chart,

tooltip = "tooltip\_text") 4

We begin by creating a date\_nice\_format variable that produces dates in the more readable *January 1, 2023* format instead of *2023-01-01* 1. We then combine this value with the state and death rate variables, saving the result as tooltip\_text 2. Next, we add a new aesthetic property in the ggplot() function 3. On its own, this doesn’t do anything until we pass the new property to the ggplotly() function 4.

Figure 9-11 shows what our new tooltip looks like: the name of the state, a nicely formatted date, and the death rate for that day.

[F09011.png]



* + - * 1. Easy-to-read interactive tooltips on the COVID death rate chart

Adding interactivity is a great way to take advantage of the medium of a website. Users who might feel overwhelmed looking at the static chart can explore the interactive version, hovering to see a summary of the results on any single day.

Hosting the Website

Now that you’ve made a website, you need a way to share it. There are various ways to do this, ranging from simple to quite complex. The easiest solution is to compress the files in your docs folder (or whatever folder you put your rendered website in) and email your ZIP file to others. They can unzip it and open the HTML files in their browser. This works fine if you know you won’t want to make changes to the data or styles of your website. But, as we discussed in Chapter 5, most projects aren’t really one-time.

In the Cloud

A better approach is to put your entire *docs* folder in a place where others can see it. This could be an internal network, Dropbox, Google Drive, Box, or something similar. Hosting the files in this way is simple to implement and allows you to give access to only those you want to see your website.

You can even automate the process of copying your *docs* folder to various online file-sharing sites using R packages: the rdrop2 package works with Dropbox, googledrive works with Google Drive, and boxr works with Box. For example, code like the following would automatically upload the project to Dropbox:

library(tidyverse)

library(rmarkdown)

library(fs)

library(rdrop2)

render\_site()

website\_files <- dir\_ls(path = "docs",

type = "file",

recurse = TRUE)

walk(website\_files, drop\_upload, path = "COVID Website")

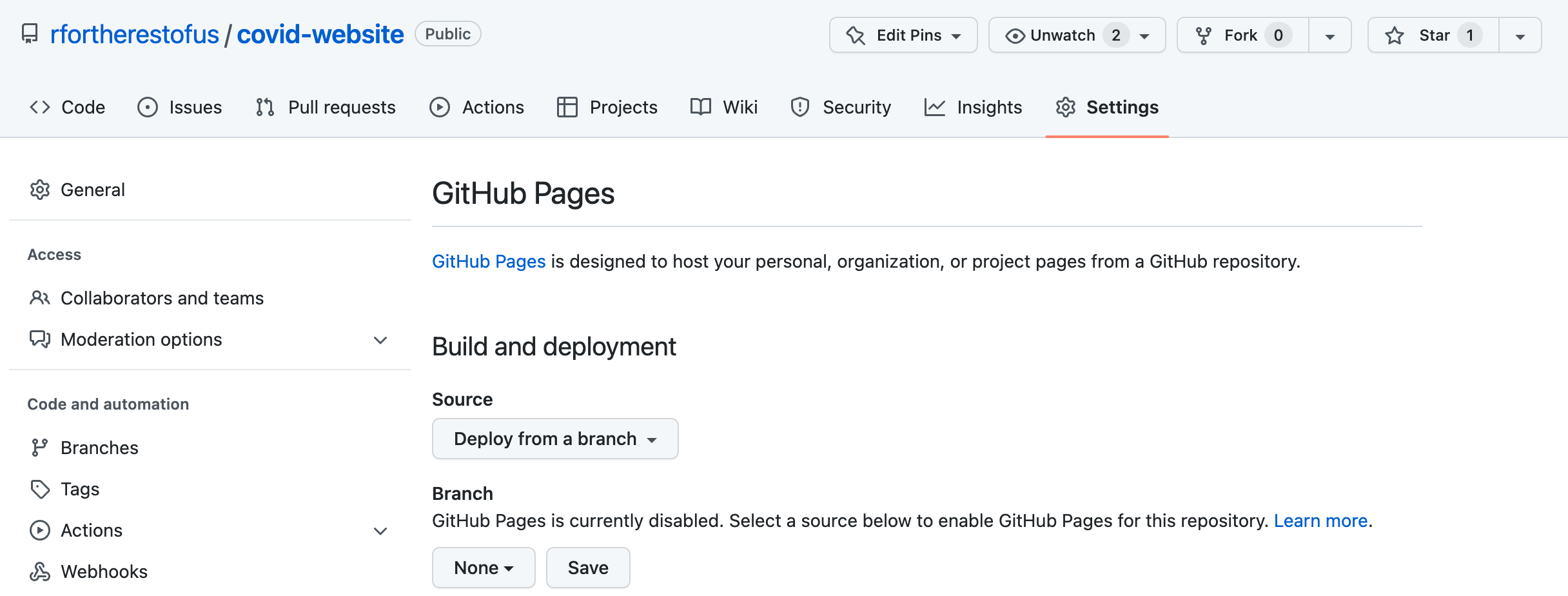
This code, which I typically add to a separate file called *render.R*, renders the site, uses the dir\_ls() function from the fs package to identify all files in the *docs* directory, and then uploads these files to Dropbox. Now you can run your entire file to generate and upload your website in one go.

Using GitHub Pages

The more complicated yet powerful solution is to use a static hosting service like GitHub Pages. This service deploys the website to a URL you’ve set up each time you commit your code to GitHub. Learning to use GitHub is an investment (the book *Happy Git and GitHub for the useR* by Jenny Bryan is a great resource), but being able to host your website for free makes it worth the time and effort.

Here’s how GitHub Pages works. Most of the time, when you look at a file on GitHub, you see its underlying source code, so if you looked at an HTML file, you’d see only the HTML code. GitHub Pages, on the other hand, shows you the rendered HTML files. To host your website on GitHub Pages, you’ll need to first push your code to GitHub. Once you have a repository set up there, go to it, then go to the Settings tab, which should look like Figure 9-12.

[F09012.png]



* + - * 1. Setting up GitHub Pages

Now choose how you want GitHub to deploy the raw HTML. The easiest approach is to keep the default source by selecting Deploy from a branch, then selecting your default branch (usually *main* or *master*). Next, select the directory containing the HTML files you want to be rendered. If you configured your website for GitHub pages at the beginning of this chapter, they should be in *docs*. Click save and wait a few minutes. GitHub should then show a URL where our website now lives.

The best part about hosting your website on GitHub Pages is that any time you update your code or data, the website will update as well. RMarkdown, distill, and GitHub Pages make building and maintaining websites a snap.

Conclusion

In this chapter, you learned to use the distill package to make websites in R. This package provides a simple way to get a website up and running with the tool you’re already using for working with data. You’ve seen how to:

* Create new pages and add them to your top navigation bar.
* Customize the look-and-feel of your website with tweaks to the CSS.
* Use wider the layouts to make content fit better on individual pages.
* Convert static data visualization and tables into interactive versions.
* Use GitHub Pages to host an always-up-to-date version of your website.

Herman has continued building websites with R. He and his colleagues at the Council of State Governments Justice Center have made a great website using Quarto, the language-agnostic version of R Markdown. This website, found at <https://projects.csgjusticecenter.org/tools-for-states-to-address-crime/>, highlights crime trends throughout the United States using many of the same techniques we’ve discussed in this chapter.

No matter whether you use distill or Quarto, using R gives you a quick way to develop complex websites without having to be a sophisticated front-end web developer. The websites look good and communicate well. They are one more way that R can help you share your results with the world.

Learn More

Consult the following resources to learn how to make websites with the distill package and to see examples of other websites made with distill:

“Building a blog with distill” by Tom Mock (2020), <https://themockup.blog/posts/2020-08-01-building-a-blog-with-distill/>

The Distillery website, showcasing websites made with distill: <https://distillery.rbind.io/>