Chapter 4

Advanced UI

Introduction

The native languages of the web are HTML (for content), CSS (for styling), and JavaScript (for behavior). Shiny is designed to be accessible for R users who aren't familiar with any of those languages. But if you do speak these languages, you can take full advantage of them with Shiny to customize your apps or extend the Shiny framework.

The previous chapter showed you the higher-level UI functions that come with Shiny. In this chapter, we'll take that a couple of steps deeper by learning how our R code gets translated into the HTML that's ultimately sent to the browser. Armed with that knowledge, you'll be able to add arbitrary HTML and CSS to your Shiny apps with ease.

If you don't know HTML, and aren't keen to learn, you can safely skip this chapter. If you don't know HTML today but may be inclined to learn it, you should be able to make sense of this chapter and understand the relationship between Shiny UI and HTML. If you are familiar with HTML already, you should find this chapter extremely useful–and you can skip over the sections that introduce HTML and CSS.

Outline

- Section 4.1 is a quick but gentle guide to the basics of HTML. It won't turn you into a web designer, but you'll at least be able to understand the rest of the chapter.
- Section 4.2 introduces tag objects, and the family of functions that produce them. These functions give us all the flexibility of writing raw HTML, while still retaining the power and syntax of R.
- Section 4.3 demonstrates several ways of incorporating custom CSS into your app (for example, to customize fonts and colors).
- Section 4.4 introduces HTML dependency objects, which can be used to tell Shiny about JavaScript/CSS dependency files that should be used.

4.1 HTML 101

To understand how UI functions in R work, let's first talk about HTML, ~~in case you're not familiar with it (or its cousin, XML)~~. ~~If~~ you've worked with HTML before, feel free to skip ~~to Section 4.2~~.

HTML is a *markup language* for describing web pages. A markup language is just a document format that contains plain text content, plus embedded instructions for annotating, or "marking up", specific sections of that content. These instructions can control the appearance, layout, and behavior of the text they mark up, and also provide structure to the document.

Here's a simple snippet of HTML:

```
This time I <em>really</em> mean it!
```

The `<em>` and `</em>` markup instructions indicate that the word `really` should be displayed with special emphasis (italics):

This time I really mean it!

`<em>` is an example of a *start tag*, and `</em>` (note the slash character) is an example of an *end tag*.

4.1.1 Inline formatting tags

`em` is just one of many HTML tags that are used to format text:

- `<strong>...</strong>` makes text bold
- `<u>...</u>` makes text underlined
- `<s>...</s>` makes text strikeout
- `<code>...</code>` makes text monospaced

4.1.2 Block tags

Another class of tags is used to wrap entire blocks of text. You can use `<p>...</p>` to break text into distinct paragraphs, or `<h3>...</h3>` to turn a line into a subheading.

```
<h3>Chapter I. Down the Rabbit-Hole</h3>

<p>Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to

<p>So she was considering in her own mind (as well as she could, for the hot day made her feel very sle
```

When rendered, this HTML looks like:

Chapter I. Down the Rabbit-Hole

Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she had peeped into the book her sister was reading, but it had no pictures or conversations in it, 'and what is the use of a book,' thought Alice 'without pictures or conversations?'

So she was considering in her own mind (as well as she could, for the hot day made her feel very sleepy and stupid), whether the pleasure of making a daisy-chain would be worth the trouble of getting up and picking the daisies, when suddenly a White Rabbit with pink eyes ran close by her.

4.1.3 Tags with attributes

Some tags need to do more than just demarcate some text. An `<a>` (for "anchor") tag is used to create a hyperlink. It's not enough to just wrap `<a>...</a>` around the link's text, as you also need to specify where the hyperlink points to.

Start tags let you include *attributes* that customize the appearance or behavior of the tag. In this case, we'll add an `href` attribute to our `<a>` start tag:

```
<p>Learn more about <strong>Shiny</strong> at <a href="https://shiny.rstudio.com">this website</a>.</p>
```

Learn more about Shiny at this website.

There are dozens of attributes that all tags accept, and hundreds of attributes that are specific to particular tags. You don't have to worry about memorizing them all–even full-time web developers don't do that. There are two attributes that are used constantly, though.

The `id` attribute uniquely identifies a tag in a document. That is, no two tags in a single document should share the same `id` value, and each tag can have zero or one `id` value. As far as the web browser is concerned, the `id` attribute is completely optional and has no intrinsic effect on the appearance or behavior of the rendered tag. However, it's incredibly useful for identifying a tag for special treatment by CSS or JavaScript, and as such, plays a crucial role for Shiny apps.

The `class` attribute provides a way of classifying tags in a document. Unlike `id`, any number of tags can have the same class, and each tag can have multiple classes (space separated). Again, classes don't have an intrinsic effect, but are hugely helpful for using CSS or JavaScript to target groups of tags.

In the following example, we've given a `<p>` tag an id and two classes:

```
<p id="storage-low-message" class="message warning">Storage space is running low!</p>
```

Storage space is running low!

Here, the `id` and `class` values have had no discernible effect. But we could, for example, write CSS that any elements with the `message` class should appear at the top of the page, and that any elements with the `warning` class should have a yellow background and bold text; and we could write JavaScript that automatically dismisses the message if the storage situation improves.

4.1.4 Parents and children

In the example above, we have a `<p>` tag that contains some text that contains `<strong>` and `<a>` tag.s We can refer to `<p>` as the *parent* of `<strong>`/`<a>`, and `<strong>`/`<a>` as the *children* of `<p>`. And naturally, `<strong>` and `<a>` are called *siblings*.

It's often helpful to think of tags and text as forming a tree structure:

```
<p>
  "Learn more about"
  <strong>
     "Shiny"
  "at"
  <a href="...">
     "this website"
  "."
```

4.1.5 Comments

Just as you can use the `#` character to comment out a line of R code, HTML lets you comment out parts of your web page. Use `<!--` to start a comment, and `-->` to end one. Anything between these delimiters will be ignored during the rendering of the web page, although it will still be visible to anyone who looks at your raw HTML by using the browser's View Source command.

```
<p>This HTML will be seen.</p>

<!-- <p>This HTML will not.</p> -->

<!--
<p>
```

```
Nor will this.
</p>
-->
```

This HTML will be seen.

4.1.6 Escaping

Any markup language like HTML, where there are characters that have special meaning, needs to provide a way to "escape" those special characters–that is, to insert a special character into the document without invoking its special meaning.

For example, the `<` character in HTML has a special meaning, as it indicates the start of a tag. What if you actually want to insert a `<` character into the rendered document–or, let's say, an entire `<p>` tag?

```
<p>In HTML, you start paragraphs with "<p>" and end them with "</p>".</p>
```

In HTML, you start paragraphs with "

" and end them with "

".

That doesn't look as we intended at all! The browser has no way of knowing that we meant the outer `<p>` and `</p>` to be interpreted as markup, and the inner `<p>` and `</p>` to be interpreted as text.

Instead, we need to escape the inner tags so they become text. The escaped version of `<` is `&lt;`, and `>` is `&gt;`.

```
<p>In HTML, you start paragraphs with "&lt;p&gt;" and end them with "&lt;/p&gt;".</p>
```

In HTML, you start paragraphs with "<p>" and end them with "</p>".

(Yes, escaped characters look pretty ugly. That's just how it is.)

Each escaped character in HTML starts with `&` and ends with `;`. There are lots of valid sequences of characters that go between, but besides `lt` (less than) and `gt` (greater than), the only one you're likely to need to know is `amp`; `&amp;` is how you insert a `&` character into HTML.

Escaping `<`, `>`, and `&` is mandatory if you don't want them interpreted as special characters; other characters can be expressed as escape sequences, but it's generally not necessary. Escaping `<`, `>`, and `&` is so common and crucial that every web framework contains a function for doing it (in our case it's `htmltools::htmlEscape`), but as we'll see in a moment, Shiny will usually do this for you automatically.

~~4.1.7 HTML tag vocabulary~~

That concludes our whirlwind tour of HTML syntax. This is all you'll need to know to follow the discussion on the pages that follow.

The much larger part of learning HTML is getting to know the actual tags that are available to you, what attributes they offer, and how they work with each other. Fortunately, there are scores of excellent, free HTML tutorials and references online; Mozilla's *Introduction to HTML* ~~is one example~~.

4.2 Generating HTML with tag objects

With this background knowledge in place, we can now talk about how to write HTML using R. To do this, we'll use the htmltools package. The htmltools package started life as a handful of functions in Shiny itself, and was later spun off as a standalone package when its usefulness for other packages–like rmarkdown and htmlwidgets–became evident.

In htmltools, we create the same trees of parent tags and child tags/text as in raw HTML, but we do so using R function calls instead of angle brackets. For example, this HTML from an earlier example:

```
<p id="storage-low-message" class="message warning">Storage space is running low!</p>
```

would look like this in R:

```
library(htmltools)
```

```
##
## Attaching package: 'htmltools'

## The following object is masked _by_ '.GlobalEnv':
##
##     knit_print.shiny.tag
```

```
p(id="storage-low-message", class="message warning", "Storage space is running low!")
```

Storage space is running low!

This function call returns a tag object; when printed at the console, it displays its raw HTML code, and when included in Shiny UI, its HTML becomes part of the user interface.

Look carefully and you'll notice:

- The `<p>` tag has become a `p()` *function call*, and the end tag is gone. Instead, the end of the `<p>` tag is indicated by the function call's closing parenthesis.
- The `id` and `class` attributes have become *named* arguments to `p()`.
- The text contained within `<p>...</p>` has become a string that is passed as an *unnamed* argument to `p()`.

Let's break down each of these bullets further.

4.2.1 Using functions to create tags

The htmltools package exports the `p` function for the `<p>` tag, but because there are scores of valid HTML tags, it doesn't export a function for each one. Only the most common HTML tags have a function directly exposed in the htmltools namespace: `<p>`, `<h1>` through `<h6>`, `<a>`, `<br>`, `<div>`, `<span>`, `<pre>`, `<code>`, `<img>`, `<strong>`, `<em>`, and `<hr>`. When writing these tags, you can simply use the tag name as the function name, e.g. `div()` or `pre()`.

~~To write all other tag~~[illegible]efix the tag name with `tags$`. For example, to create a `<ul>` tag, there's no dedicated `ul()` functi[illegible]ut you can call `tags$ul()`. The `tags` object is a named list that htmltools provides, and it comes preloaded with almost all of the valid tags in the HTML5 standard.

When writing a lot of HTML from R, you may find it tiresome to keep writing `tags$`. If so, you can use the `withTags` function to wrap an R expression, wherein you can omit the `tags$` prefix. In the following code, we call `ul()` and `li()`, whereas these would normally be `tags$ul()` and `tags$li()`.

```
withTags(
  ul(
    li("Item one"),
    li("Item two")
  )
)
```

Item one

Item two

Finally, in some relatively obscure cases, you may find that not even `tags` supports the tag you have in mind; this may be because the tag is newly added to HTML and has not been incorporated into htmltools yet, or because it's a tag that isn't defined in HTML per se but is still understood by browsers (e.g. the `<circle>` tag from SVG). In these cases, you can fall back to the `tag()` (singular) function and pass it any tag name.

```
tag("circle", list(cx="10", cy="10", r="20", stroke="blue", fill="white"))
```

(Notice that the `tag()` function alone needs its attribute and children wrapped in a separate `list()` object. This is a historical quirk, don't read into it.)

4.2.2 Using named arguments to create attributes

When calling a tag function, any named arguments become HTML attributes.

```
# From https://getbootstrap.com/docs/3.4/javascript/#collapse
a(class="btn btn-primary", `data-toggle`="collapse", href="#collapseExample",
  "Link with href"
)
```

```
## <a class="btn btn-primary" data-toggle="collapse" href="#collapseExample">Link with href</a>
```

The preceding example includes some attributes with hyphens in their names. Be sure to quote such names using backticks, or single or double quotes. Quoting is also permitted, but not required, for simple alphanumeric names.

Generally, HTML attribute values should be single-element character vectors, as in the above example. Other simple vector types like integers and logicals will be passed to `as.character()`.

Another valid attribute value is `NA`. This means that the attribute should be included, but without an attribute value at all:

```
tags$input(type = "checkbox", checked = NA)
```

```
## <input type="checkbox" checked/>
```

You can also use `NULL` as an attribute value, which means the attribute should be ignored (as if the attribute wasn't included at all). This is helpful for conditionally including attributes.

```
is_checked <- FALSE
tags$input(type = "checkbox", checked = if (is_checked) NA)
```

```
## <input type="checkbox"/>
```

4.2.3 Using unnamed arguments to create children

Tag functions interpret unnamed arguments as children. Like regular HTML tags, each tag object can have ro, one, or more children; and each child can be one of several types of objects:

4.2.3.1 Tag objects

Tag objects can contain other tag objects. Trees of tag objects can be nested as deeply as you like.

```
div(
  p(
    strong(
```

```r
      a(href="https://example.com", "A link")
    )
  )
)
```

A link

4.2.3.2 Plain text

Tag objects can contain plain text, in the form of single-element character vectors.

```r
p("I like turtles.")
```

I like turtles.

Note that character vectors of longer length are not supported. Use the `paste` function to collapse such vectors to a single element.

```r
str(LETTERS)
```

```
##  chr [1:26] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" ...
```

```r
div(paste(LETTERS, collapse = ", "))
```

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z

One important characteristic of plain text is that htmltools assumes you want to treat all characters as *plain* text, including characters that have special meaning in HTML like `<` and `>`. As such, any special characters will be automatically escaped:

```r
div("The <strong> tag is used to create bold text.")
```

The <strong> tag is used to create bold text.

4.2.3.3 Verbatim HTML

Sometimes, you may have a string that should be interpreted as HTML; similar to plain text, except that special characters like `<` and `>` should retain their special meaning and be treated as markup. You can tell htmltools that these strings should be used verbatim (not escaped) by wrapping them with `HTML()`:

```r
html_string <- "I just <em>love</em> writing HTML!"
div(HTML(html_string))
```

I just love writing HTML!

Warning: Be very careful when using the `HTML()` function! If the string you pass to it comes from an untrusted source, either directly or indirectly, it could compromise the security of your Shiny app via an extremely common type of security vulnerability known as Cross-Site Scripting (XSS).

For example, you may ask a user of your app to provide their name, store that value in a database, and later display that name to a different user of your app. If the name is wrapped in `HTML()`, then the first user could provide a fragment of malicious HTML code in place of their name, which would then be served by Shiny to the second user. While I'm not aware of any such attacks having ever happened with a Shiny app, similar attacks have been carried out against companies as tech-savvy as Facebook, eBay, and Microsoft.

It is safe to use `HTML()` if your Shiny app will only ever be run locally for a single user (not deployed on a server for multiple users), or if you are absolutely sure you know where the HTML string came from and that its contents are safe.

4.2.3.4 Lists

While each call to a tag function can have as many unnamed arguments as you want, you can also pack multiple children into a single argument by using a `list()`. The following two code snippets will generate identical results:

```r
tags$ul(
  tags$li("A"),
  tags$li("B"),
  tags$li("C")
)
```

A

B

C

```r
tags$ul(
  list(
    tags$li("A"),
    tags$li("B"),
    tags$li("C")
  )
)
```

A

B

C

It can sometimes be handy to use a list when generating tag function calls programmatically. The snippet below uses `lapply` to simplify the previous example:

```r
tags$ul(
  lapply(LETTERS[1:3], tags$li)
)
```

A

B

C

4.2.3.5 NULL

You can use `NULL` as a tag child. `NULL` children are similar to `NULL` attributes; they're simply ignored, and are only supported to make conditional child items easier to express.

In this example, we use `show_beta_warning` to decide whether or not to show a warning; if not, the result of the `if` clause will be `NULL`.

```r
show_beta_warning <- FALSE

div(
  h3("Welcome to my Shiny app!"),
  if (show_beta_warning) {
    div(class = "alert alert-warning", role = "alert",
      "Warning: This app is in beta; some features may not work!"
    )
  }
)
```

Welcome to my Shiny app!

4.2.3.6 Mix and match

Tag functions can be called with any number of unnamed arguments, and different types of children can be used within a single call.

```
div(
  "Text!",
  strong("Tags!"),
  HTML("Verbatim <span>HTML!</span>"),
  NULL,
  list(
    "Lists!"
  )
)
```

Text! Tags! Verbatim HTML! Lists!

4.3 Customizing with CSS

In the previous section, we talked about HTML, the markup language that specifies the structure and content of the page. Now we'll talk about Cascading Style Sheets (CSS), the language that specifies the visual style and layout of the page. Similar to our treatment of HTML, we'll give you an extremely superficial introduction to CSS—just enough to be able to parse the syntax visually, not enough to write your own. Then we'll talk about the mechanisms available in Shiny for adding your own CSS.

4.3.1 Introduction to CSS

As we saw in the previous sections, we use HTML to create a tree of tags and text. CSS lets us specify directives that control how that tree is rendered; each directive is called a *rule*.

Here's some example CSS that includes two rules: one that causes all `<h3>` tags (level 3 headings) to turn red and italic, and one that hides all `<div class="alert">` tags in the `<footer>`:

```
h3 {
  color: red;
  font-style: italic;
}

footer div.alert {
  display: none;
}
```

The part of the rule that precedes the opening curly brace is the *selector*; in this case, `h3`. The selector indicates which tags this rule applies to.

The parts of the rule inside the curly braces are *properties*. This particular rule has two properties, each of which is terminated by a semicolon.

4.3.1.1 CSS selectors

The particular selector we used in ~~these cases~~ (`h3` and `footer div.alert`) are very simple, but selectors can be quite complex. You can select tags that match a specific `id` or `class`, select tags based on their parents, select tags based on whether they have sibling tags. You can combine such critieria together using “and”, “or”, and “not” semantics.

Here are some extremely common selector patterns:

- `.foo` - All tags whose `class` attributes include `foo`
- `div.foo` - All `<div>` tags whose `class` attributes include `foo`
- `#bar` - The tag whose `id` is `bar`
- `div#content p:not(#intro)` - All `<p>` tags inside the `<div>` whose `id` is `content`, except the `<p>` tag whose `id` is `intro`

A full introduction to CSS selectors is outside the scope of this chapter, but if you’re interested in learning more, the Mozilla tutorial on CSS selectors is a good place to start.

4.3.1.2 CSS properties

The syntax of CSS properties is very simple and intuitive. What’s challenging about CSS properties is that there are so darn many of them. There are dozens upon dozens of properties that control typography, margin and padding, word wrapping and hyphenation, sizing and positioning, borders and shadows, scrolling and overflow, animation and 3D transforms… the list goes on and on.

Here are some examples of common properties:

- `font-family: Open Sans, Helvetica, sans-serif;` Display text using the Open Sans typeface, if it’s available; if not, fall back first to Helvetica, and then to the browser’s default sans serif font.
- `font-size: 14pt;` Set the font size to 14 point.
- `width: 100%; height: 400px;` Set the width to 100% of the tag’s container, and the height to a fixed 400 pixels.
- `max-width: 800px;` Don’t let the tag grow beyond 800 pixels wide.

Most of the effort in mastering CSS is in knowing what properties are available to you, and understanding when and how to use them. Again, the rest of this large topic is outside of our scope, but you can learn more from Mozilla’s CSS resources page.

4.3.2 Including custom CSS in Shiny

Shiny gives you several options for adding custom CSS into your apps. Which method you choose will depend on the amount and complexity of your CSS.

####Inline style tag with `tags$style`

If you have one or two very simple rules, the easiest way to add CSS is by inserting a `<style>` tag, using `tags$style()`. This can go almost anywhere in your UI.

```
ui <- fluidPage(
  tags$style(HTML("
    body, pre { font-size: 18pt; }
  "))
)
```

The `HTML()` is only strictly necessary if your CSS code includes special characters like `<`, `>`, or `&`; those characters will be escaped by default, which will lead to invalid CSS. I like to add it as a matter of habit, though you should heed the warning about untrusted input in Section 4.2.3.3.

4.3.2.1 Standalone CSS file with `includeCSS`

~~The~~ second method is to write a standalone `.css` file, and use the `includeCSS` function to add it to your UI. If you're writing more than a couple of lines of CSS, this gives you the benefit of being able to use a text editor that knows how to interpret CSS. Most text editors, including RStudio IDE, will be able to provide syntax highlighting and autocompletion when editing `.css` files.

For example, if you have a `custom.css` file sitting next to your `app.R`, you can do something like this:

```
ui <- fluidPage(
  includeCSS("custom.css"),
  ... # the rest of your UI
)
```

The `includeCSS` call will return a `<style>` tag, whose body is the content of `custom.css`.

If you wish for this CSS to be hoisted into the page's `<head>` tag, you can call `tags$head(includeCSS("custom.css"))`, though in practice it doesn't make a lot of difference either way.

4.3.2.2 Standalone CSS file with `<link>` tag

Perhaps you have a large amount of CSS in a standalone `.css` file, and you don't want it to be inserted directly into the HTML page, as `includeCSS` does. You can also choose to serve up the `.css` file at a separate URL, and link to it from your UI.

To do this, create a `www` subdirectory in your application directory (the same directory that contains `app.R`) and put your CSS file there—for example, `www/custom.css`. Then add the following line to your UI:

```
tags$head(tags$link(rel="stylesheet", type="text/css", href="custom.css"))
```

(Note that the `href` attribute should *not* include `www`; Shiny makes the contents of the `www` directory available at the root URL path.)

This approach makes sense for CSS that is intended for a specific Shiny app. If you are writing CSS for a reusable component, especially one in an R package, keep reading to learn about HTML dependencies.

4.4 Managing JavaScript/CSS dependencies

One of the best things about working with HTML is the absolutely mindboggling number of open source JavaScript and/or CSS libraries that you can incorporate into your pages. Such libraries can be as simple as a few dozen styling rules in a single .css file, or as complex as 670KB of minified JavaScript for visualizing spatial data using WebGL.

Whether simple or complex, the first step in using any JavaScript/CSS library is referencing its .js and/or .css files from your HTML. Any well-documented library will have a "Getting Started" section that includes a snippet of HTML, including `<script>` and/or `<link>` tags, that need to be pasted into the special `<head>` section of the webpage.

This puts the onus on the webpage author to know about all the JavaScript/CSS dependencies that are used by any element of the page. For traditional web developers, that's totally normal. But for Shiny (and R Markdown, and htmlwidgets), we wanted to free R users from having to think very much about JavaScript/CSS dependencies at all.

For example, the `sliderInput` that comes with Shiny happens to be implemented using a JavaScript library called ionRangeSlider. With the traditional approach, if you wanted to add a `sliderInput` to your Shiny app, you'd also have to make a separate declaration somewhere in your UI to load the .js and .css files for ionRangeSlider. And you'd have to make sure that you only make that declaration once per page, lest you accidentally load a library twice, which can cause some libraries to stop working.

For Shiny, we wanted instead to have R users simply call `sliderInput()`, and let ~~us~~ sort out the necessary dependencies automatically. In fact, we want R users to be able to combine whatever UI objects they want, and just not think about dependencies at all.

To make this possible, we created a first-class object for HTML dependencies.

An HTML dependency object is a description of a single JavaScript/CSS library. As a Shiny app author, you generally don't create these directly. Instead, these objects are created by package authors who want to make reusable UI functions that depend on external JavaScript/CSS—like `sliderInput()`. If you're such a package author, you absolutely should be using HTML dependency objects rather than calling `tags$link()`, `tags$script()`, or `includeCSS` directly.

4.4.1 Creating HTML dependency objects

To form such an object, you call `htmlDependency` with the following arguments:

1. `name`: The name of the library.
2. `version`: The version number of the library. (If two or more HTML dependency objects are found with the same name but different version numbers, only the one with the latest version number will be loaded.)
3. `src`: A single location where the library's resources (JavaScript, CSS, and/or images) can be found. This is usually an absolute filesystem path, but can also be a web URL. For resources that are bundled into an R package, you can provide a relative filesystem path (relative to the installed package directory, i.e. `system.file(package="pkgname")`) if you provide an additional `package` argument.
4. `script`: Relative paths to JavaScript files that should be loaded.
5. `stylesheet`: Relative paths to CSS files that should be loaded.

Here's an example from the leaflet.mapboxgl package:

```
mapbox_gl_dependency <- htmlDependency(
  "mapbox-gl",
  "0.53.1",
  src = "node_modules/mapbox-gl/dist",
  package = "leaflet.mapboxgl",
  script = "mapbox-gl.js",
  stylesheet = "mapbox-gl.css",
  all_files = FALSE
)
```

In this example, the leaflet.mapboxgl package has a `node_modules/mapbox-gl/dist` subdirectory that contains two files: `mapbox-gl.js` and `mapbox-gl.css`.

For this particular library, we could also have pointed to these hosted URLs:

https://api.tiles.mapbox.com/mapbox-gl-js/v0.53.1/mapbox-gl.js https://api.tiles.mapbox.com/mapbox-gl-js/v0.53.1/mapbox-gl.css

We can do this by passing `src=c(href="https://api.tiles.mapbox.com/mapbox-gl-js/v0.53.1/")` and removing the `package` argument. However, be aware that a small but significant number of Shiny users do so from networks that are disconnected from the wider Internet for security reasons, and for those users, this dependency would fail to load.

4.4.2 Using HTML dependency objects

Once you've created an HTML dependency object, using it is straightforward.

If you have a function that returns a tag object, have that function return a tag object and dependency instead, using a `tagList`.

```r
create_mapbox_map <- function(map_id) {
  tagList(
    div(id = map_id),
    mapbox_gl_dependency
  )
}
```

You can bundle any number of dependencies with your HTML this way; just add additional dependency arguments to the `tagList`.

TODO: Exercises?