Outstanding User Interfaces with Shiny

David Granjon

2020-05-22

Contents

Prerequisites

- Be familiar with Shiny
- Basic knowledge in HTML and JavaScript is a plus but not mandatory

Disclaimer

This book is not an HTML/Javascript/CSS course! Instead, it provides a *survival kit* to be able to customize Shiny. I am sure however that readers will want to explore more about these topics.

Is this book for me?

You should read this book if you answer yes to the following questions:

- Do you want to know how to develop outstanding shiny apps?
- Have you ever wondered how to develop new input widgets?

Related content

See the RStudio Cloud dedicated project.

```
library(shiny)
library(shinydashboard)
library(shiny.semantic)
library(cascadess)
library(htmltools)
library(purrr)
library(magrittr)
library(ggplot2)
library(thematic)
```

Chapter 1

Introduction

There are various Shiny focused resources introducing basic as well as advanced topics such as modules and Javascript/R interactions, however, handling advanced user interfaces design was never an emphasis. Clients often desire custom templates, yet this generally exceeds core features of Shiny (not out of the box).

Generally, R App developers lack a significant background in web development and often find this requirement overwhelming. It was this sentiment that motivated writing this book, namely to provide readers the necessary knowledge to extend Shiny's layout, input widgets and output elements. This project officially started at the end of 2018 but was stopped when Joe Cheng revealed the upcoming Mastering Shiny Book. Fortunately, the later, does not cover the customization of Shiny user interfaces. Besides, this book may constitute a good complement to the work in progress Engineering Production-Grade Shiny Apps by the ThinkR team, where the link between Shiny and CSS/JavaScript is covered.

This book is organized into four parts.

- We first go through the basics of HTML, JavaScript and jQuery and finish with a chapter dedicated to the partially hidden features of Shiny, yet so fun
- In part 2, we dive into the `{htmltools}` package, providing functions to create and manipulate shiny tags as well as manage dependencies
- Part 3 focuses on the development of a new template for Shiny by demonstrating examples from the `{tablerDash}`, `{bs4Dash}` and `{shinyMobile}` packages. These, and more may be explored further as part of the RinteRface project.
- Part 4 present some tools of the R community, like `{fresh}`, to beautify apps with only few lines of code

Survival Kit

This part will give you basis in HTML, JavaScript to get started...

Chapter 2

HTML

This chapter provides a short introduction to the HTML language. As a quick example, open up RStudio and perform the following:

- Load shiny with `library(shiny)`
- Execute `p("Hello World")`

Notice the output format is an example of an HTML tag!

2.1 HTML Basics

HTML (Hypertext Markup Language) is derived from SGML (Standard Generalized markup Language). An HTML file contains tags that may be divided into 2 catagories:

- paired-tags: the text is inserted between the opening and the closing tag
- closing-tags

```
<!-- paired-tags -->
<p></p>
<div></div>

<!-- self-closing tags -->
<iframe/>
<img/>
<input/>
<br/>
```

Tags may be divided into 3 categories, based on their role:

- structure tags: they consitute the skeleton of the HTML page (`<title></title>`, `<head></head>`, `<body></body>`)
- control tags: script, inputs and buttons (and more). Their role is to include external ressources, provide interactivity with the user
- formatting tags: to control the size, font of the wrapped text

Finally, we distinguish block and inline elements:

- block elements may contain other tags and take the full width (block or inline). `<div></div>` is the most commonly used block element. All elements of a block are printed on top of each others
- inline elements (for instance `<span></span>`, `<a></a>`) are printed on the same line. They can not contain block tags but may contain other nested inline tags. In practice, we often see `<a><span></span></a>`
- inline-block elements allow to insert block element in an inline

Consider the following example. This is clearly a bad use of HTML conventions since an inline tag can not host block elements.

```
<span>
  <div><p>Hello World</p></div>
  <div></div>
</span>
```

Importantly, `<div>` and `<span>` don't have any semantic meaning, contrary to `<header>` and `<footer>`, which allow to structure the HTML page.

2.2 Tag attributes

Attributes are text elements allowing to specify some properties of the tag. For instance for a link tag (`<a></a>`), we actually expect more than just the tag itself: a target url and how to open the new page … In all previous examples, tags don't have any attributes. Yet, there exist a large range of attributes and we will only see 2 of them for now (the reason is that these are the most commonly used in CSS and JavaScript):

- class: may be shared between multiple tags
- id: each must be unique

```
<div class="awesome-item" id="myitem"></div>
<!-- the class awesome-item may be applied to multiple tags -->
<span class="awesome-item"></span>
```

Both attributes are widely used by CSS and JavaScript (see Chapter 3 with the jQuery selectors) to apply a custom style to a web page. Class attributes apply to multiple elements, however the id attribute is restricted to only one item.

Interestingly, there exists another attribute category, know as non-standard attributes like `data-toggle`. We will see them later in the book (see Chapter 10).

2.3 HTML page: skeleton

An HTML page is a collection of tags which will be interpreted by the web browser step by step. The simplest HTML page may be defined as follows:

```
<!DOCTYPE HTML>
<html>
  <head>
  <!-- head content here -->
  </head>
  <body>
  <!-- body content here -->
  </body>
</html>
```

- `<html>` is the may wrapper
- `<head>` and `<body>` are the 2 main children
  - `<head>` contains dependencies like styles and JavaScript files (but not only),
  - `<body>` contains the page content and it is displayed on the screen. We will see later that JavaScript files are often added just before the end of the `<body>`.

Only the body content is displayed on the screen!

Let's write the famous Hello World in HTML:

```
<!DOCTYPE HTML>
<html>
  <head>
  <!-- head content here -->
  </head>
  <body>
  <p>Hello World</p>
  </body>
</html>
```

In order to preview this page in a web browser, you need to save the above snippet to a script `hello-world.html` and double-click on it. It will open with your default web browser.

2.4 About the Document Object Model (DOM)

The DOM stands for "Document Object Model" and is a convenient representation of the html document. There actually exists multiple DOM types, namely DOM-XML and DOM-HTML but we will only focus on the latter. If we consider the last example (Hello World), the associated DOM tree may be inspected in Figure 2.1.

2.4.1 Visualizing the DOM: the HTML inspector

Below, we introduce a tool that will facilitate our exploration of beautiful shiny user interfaces. In this section, we restrict the description to the first panel of the HTML inspector [1]. This feature is available in all web browser, however for demonstration purposes, we will only focus on Chrome.

- Open the hello-world.html example in a web browser (google chrome here)
- Right-click to open the HTML inspector (developer tools must be enabled if it is not the case)

The HTML inspector is a convenient tool to explore the structure of the current HTML page. On the left-hand side, the DOM tree is displayed where we clearly see that `<html>` is the parent of `<head>` and `<body>`. `<body>` has also 1 child, that is `<p>`. We may preview any style (CSS) associated to the selected element on the right panel as well as Event Listeners (JavaScript), which will be discussed that in the next chapter.

2.5 Preliminary introduction to CSS and JavaScript

To introduce this section, I propose to look at the very first website, early in the 90's (August 1991 exactly). From an esthetic point of view (see Figure 2.2), this is far from what we can observe today as shown in Figure 2.3.

How can we explain that difference? One of the main reason is the absence of CSS (Cascading Style Sheet) since the first CSS release only appeared in

[1] As shown in Figure 2.1, the inspector also has tools to debug JavaScript code, inspect files, run performances audit, … We will describe some of these later in the book.

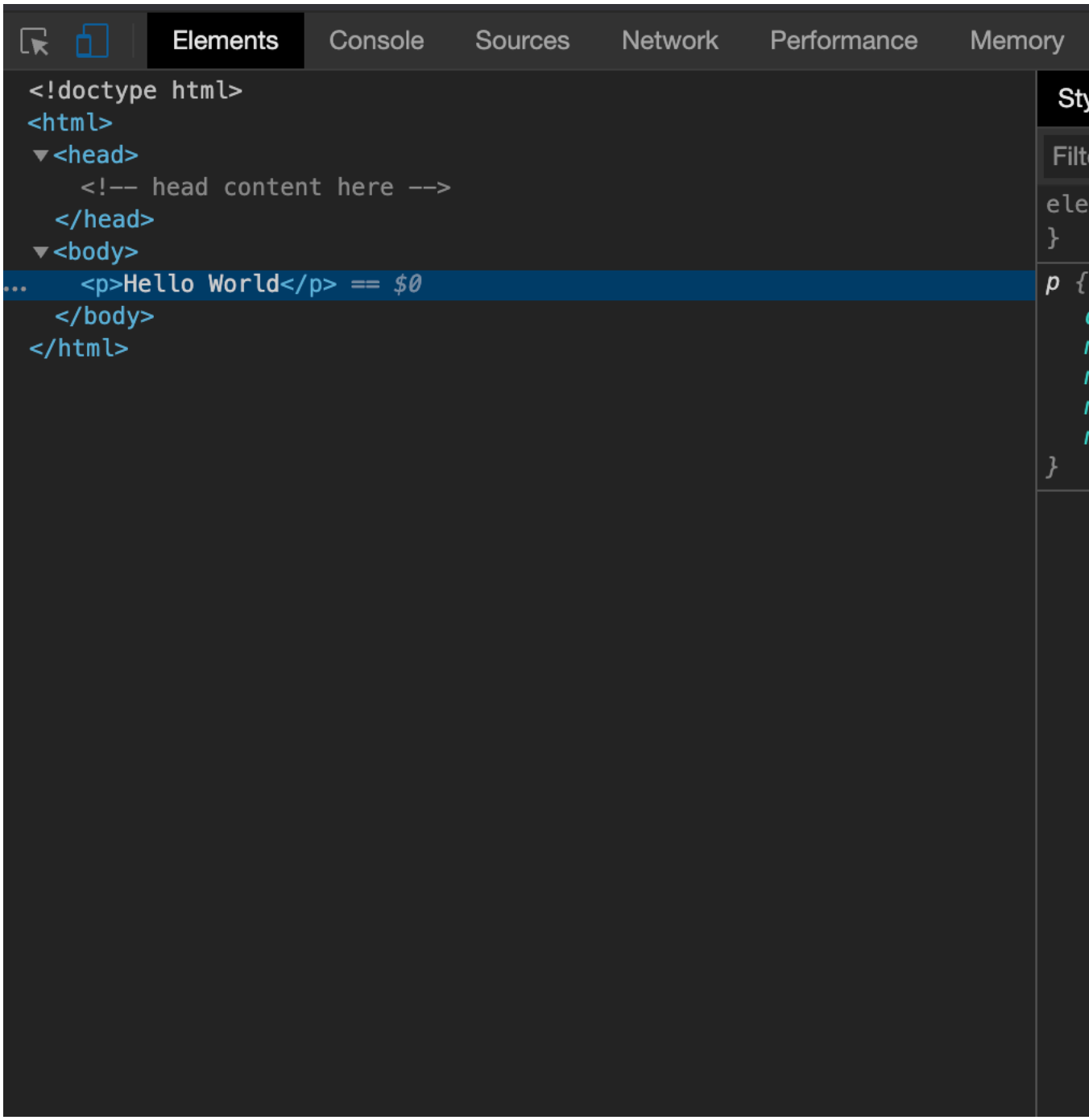


Figure 2.1: Inspection of the DOM in the Hello World example

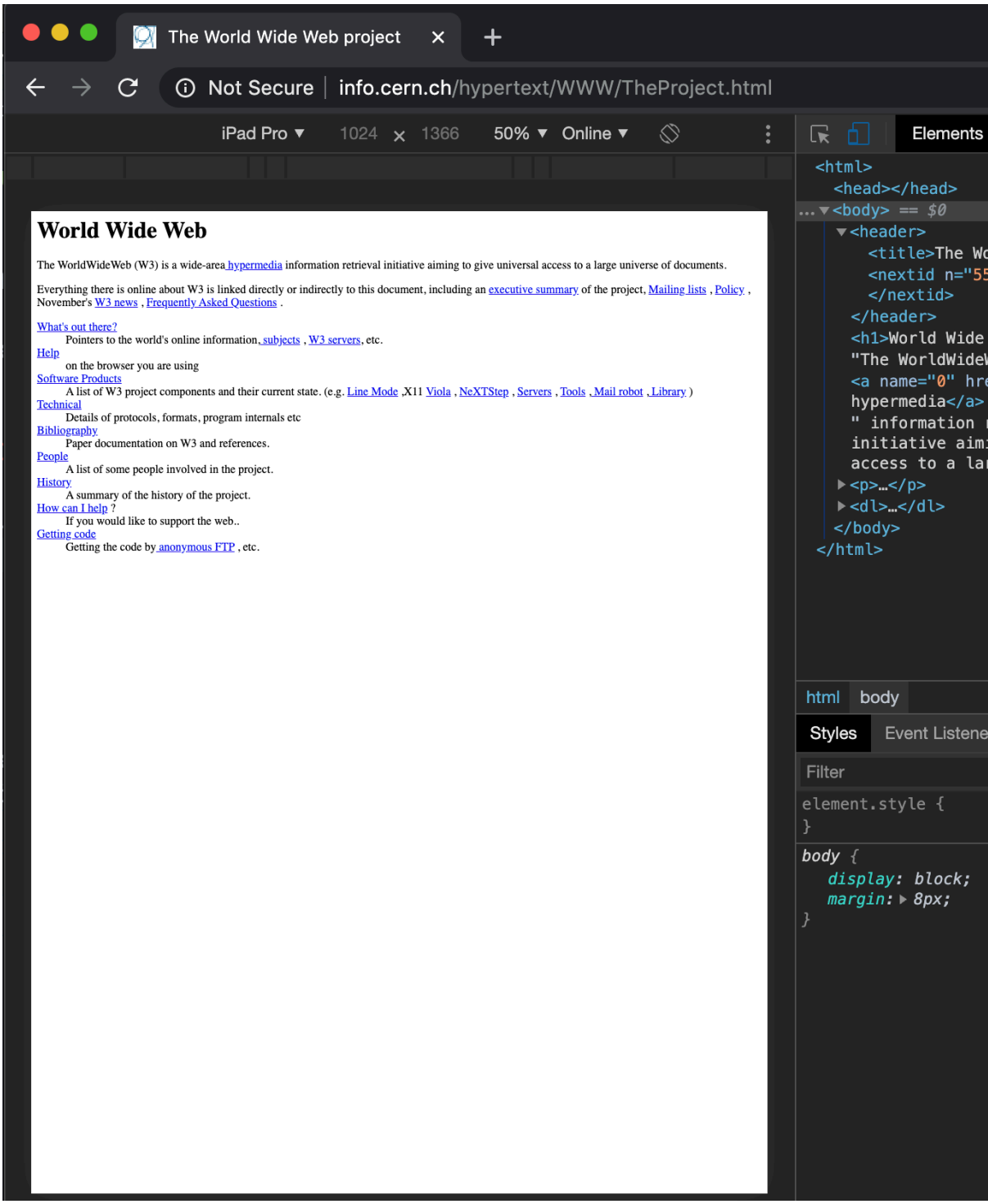


Figure 2.2: World wide web website

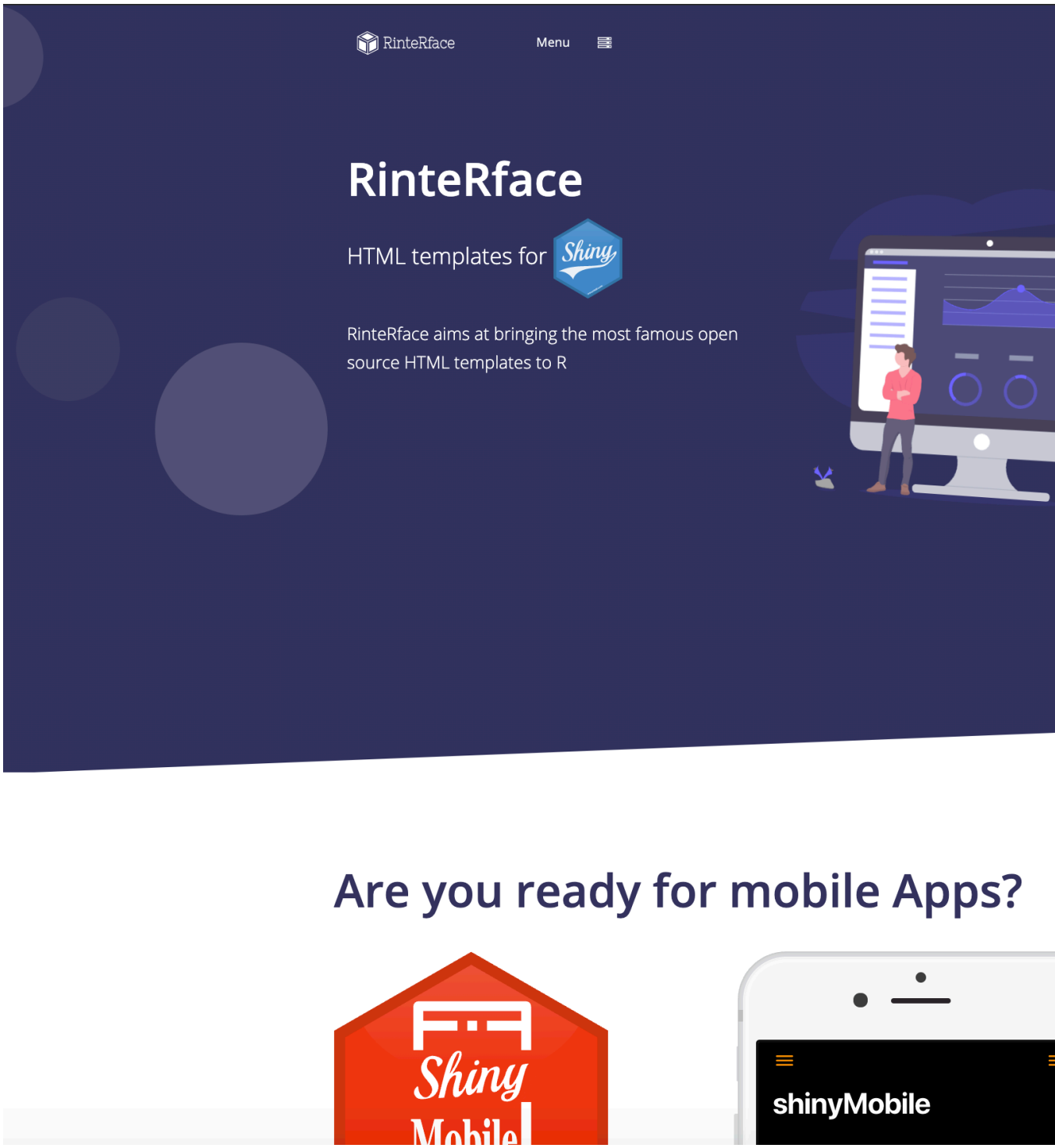


Figure 2.3: RinteRface website: https://rinterface.com

December 1996, that is 5 years later than the first web site publication. CSS allows to deeply customize the appearance of any web page by changing colors, fonts, margins and much more. We acknowledge, the role of JavaScript cannot be demonstrated through the previous example. Yet its impact is as important as CSS, so that it is now impossible to dissociate HTML, CSS and JavaScript.

2.5.1 HTML and CSS

CSS (Cascading Style Sheets) changes the style of HTML tags by targeting specific classes or ids. For instance, if we want all p tags to have red color we will use:

```
p {
  color: red;
}
```

To include CSS in an HTML page, we use the `<style>` tag as follows:

```
<!DOCTYPE HTML>
<html>
  <head>
    <style type="text/css">
      p {
        color: red;
      }
    </style>
  </head>
  <body>
  <p>Hello World</p>
  </body>
</html>
```

You may update the hello-world.html script and run it in your web-browser to see the difference. The example may be slight, but shows how we may control the look and feel of the display. In a development context, we will see later that css files may so big that it is better to include them in external files.

2.5.2 HTML and JavaScript

You will see how quickly/seamlessly you may add awesome features to your shiny app.

Let's consider the following example:

```
<!DOCTYPE HTML>
<html>
  <head>
    <style type="text/css">
      p {
        color: red;
      }
    </style>
    <script language="javascript">
      // displays an alert
      alert('Click on the Hello World text!');
      // change text color
      function changeColor(color){
        document.getElementById('hello').style.color = "green";
      }
    </script>
  </head>
  <body>
    <!-- onclick attributes applies the JavaScript function changeColor define above -->
    <p id="hello" onclick="changeColor('green')">Hello World</p>
  </body>
</html>
```

In few lines of code, you can change the color of the text. This is only the beginning!

Let's move to the next chapter to discover JavaScript!

Chapter 3

JavaScript

3.1 Introduction

JavaScript (JS) was created in 1995 by Brendan Eich and is also known as ECMAScript (ES). Interestingly, you might have heard about ActionScript, which is no more than an implementation of ES by Adobe Systems. Nowadays, JavaScript is the centerpiece of web development across all websites.

Here is a quick example. If you have a personal blog, you probably know Hugo or Jekyll. These tools allow one to rapidly develop a professionnal looking (or at least not too ugly) blog in just a few minutes. This allows bloggers to focus on the content, which is really the point! Now, if you open the HTML inspector introduced in Chapter 2, click on the elements tab, which may open by default, and uncollapse the `<head>` tag, you see that a lot of scripts are included, as shown in Figure 3.1. Similarly for the `<body>` tag.

There are 2 ways to include scripts:

- Use the `<script>` tag with the JS code inside
- Add the `onclick` attribute to a button to trigger JS as soon as it is clicked (This is similar to event listeners, see below)
- Import an external file containing the JS code and only

```
<script type="text/javascript">
// JS code here
</script>
```

```
<!-- We use the src attribute to link the external file -->
<script type="text/javascript" src="file.js">
```

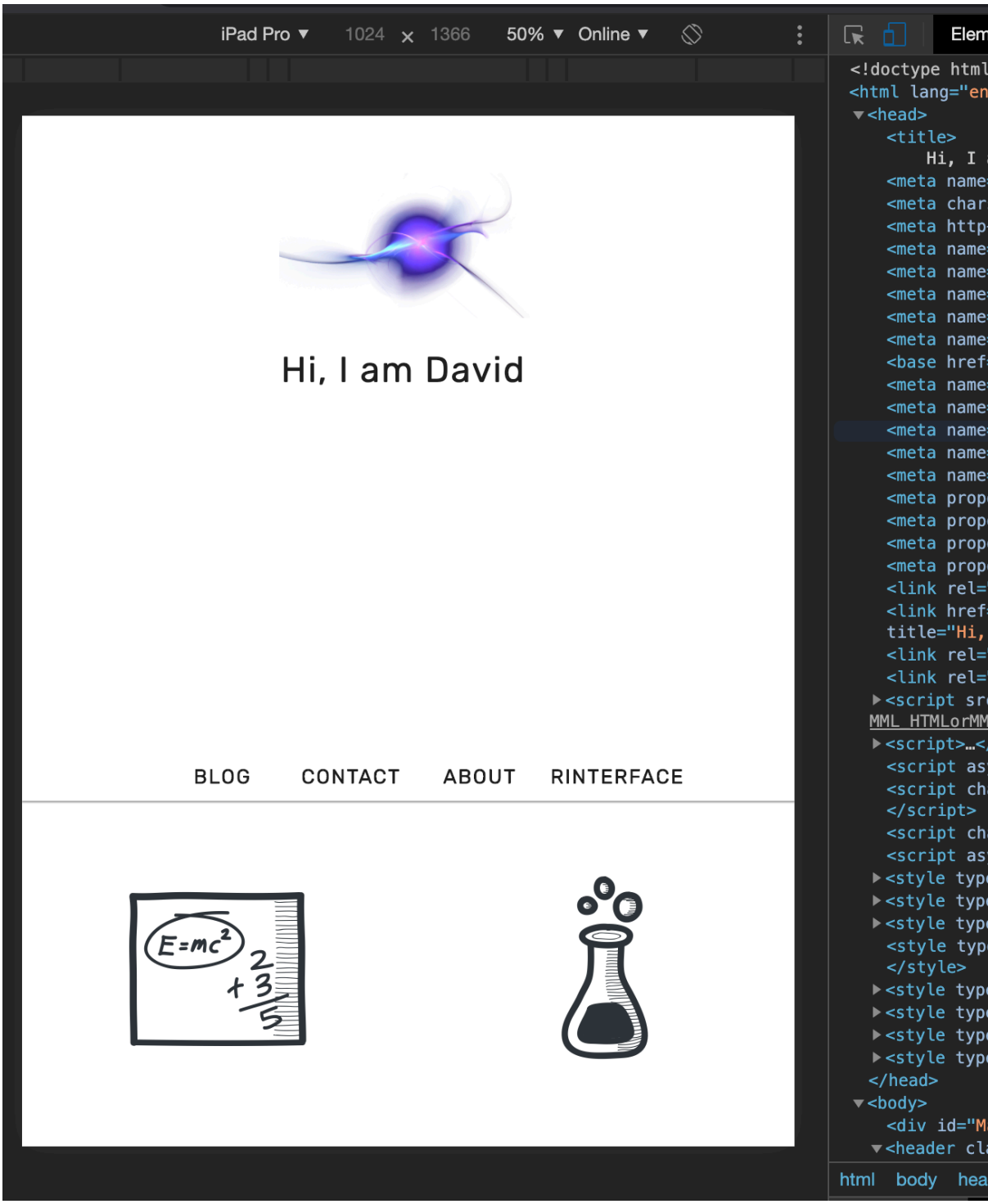


Figure 3.1: A website is full of JavaScript

Whether to choose the first, second or third method depends on the content of your script. If we consider the JS library jQuery, it unfortunately contains so much code making it a challenge to understand. This often makes users avoid the first method.

3.2 Setup

Like R or Python, JavaScript (JS) is an interpreted language, executed client-side, in other words in the browser. This also means that JS code may not be run without a suitable tool.

3.2.1 Node

Node contains an interpreter for JS as well as a dependencies manager, npm (Node Package Manager). To install Node on your computer, browse to the website and follow the installation instructions. Afterwards, open a terminal and check if

```
$ which node
$ node --version
```

returns something. If not, Node may not be properly installed.

3.2.2 Choose a good IDE

Personally, I really like VSCode for coding with JS, as it contains a Node interpreter allowing you to seamlessly execute any JS code. As a side note, I encourage you to try the dracula color theme, which is my favorite! Many also chose the Rstudio IDE, provided that you have Node installed. Below, we will see how to run a JS code in both IDE's.

3.2.3 First Script

Let's write our first script:

```
console.log("Hello World");
```

You notice that all instruction end by `;`. You can run this script either in Rstudio IDE or VSCode.

In VSCode, clicking on the run arrow (top center) of Figure 3.2, triggers the `node hello.js` command, which tells Node to run the script. We see the result

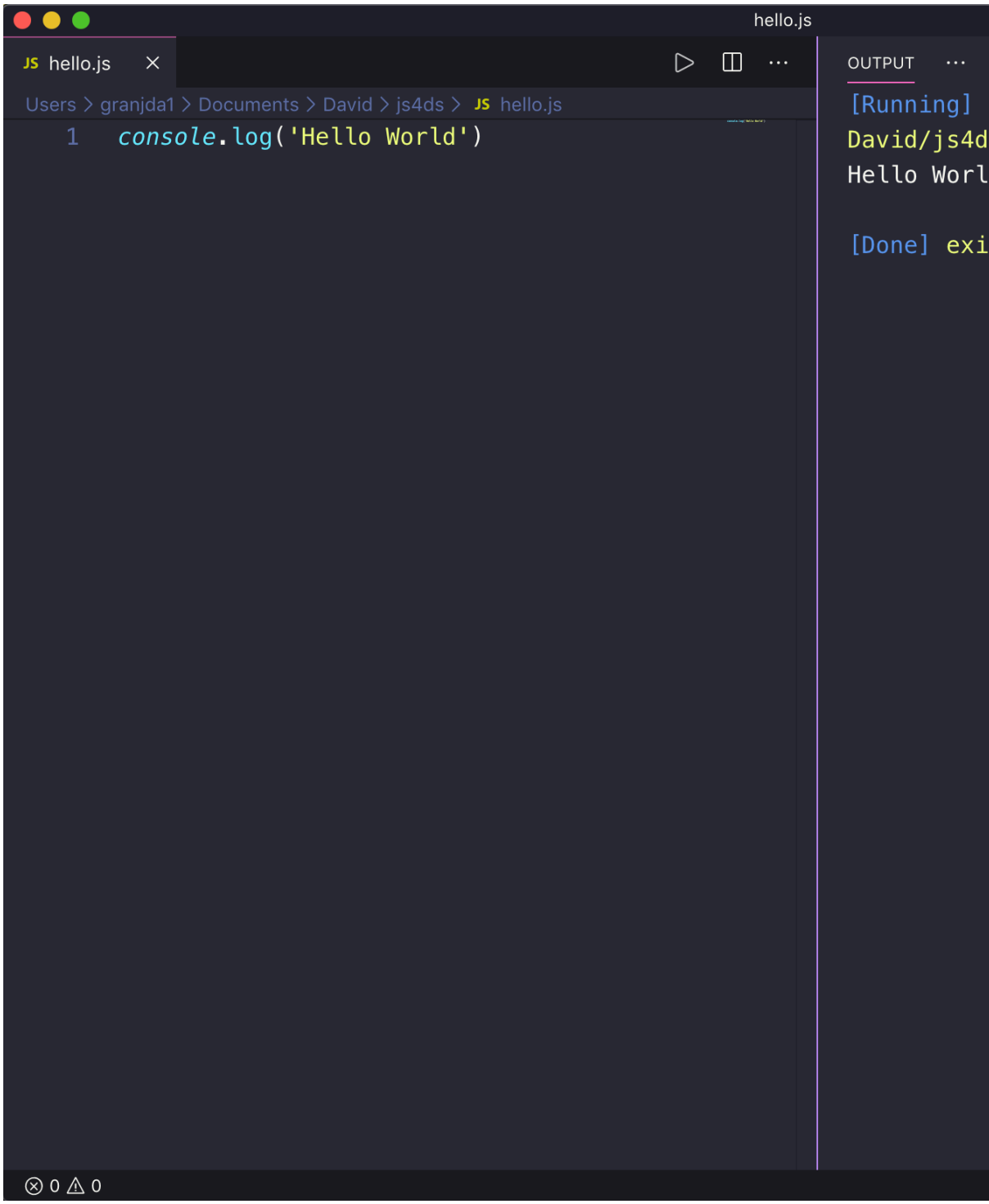


Figure 3.2: Run JS in VSCode

in the right panel (code=0 means the execution is fine and we even have the compute time). To run this script in the RStudio IDE, one needs to click on the terminal tab (you could also open a basic terminal) and type `node hello.js` (or `node mycustompath/hello.js` if you are not in the folder containing the script). You should see the Hello World message in the console (see Figure 3.3).

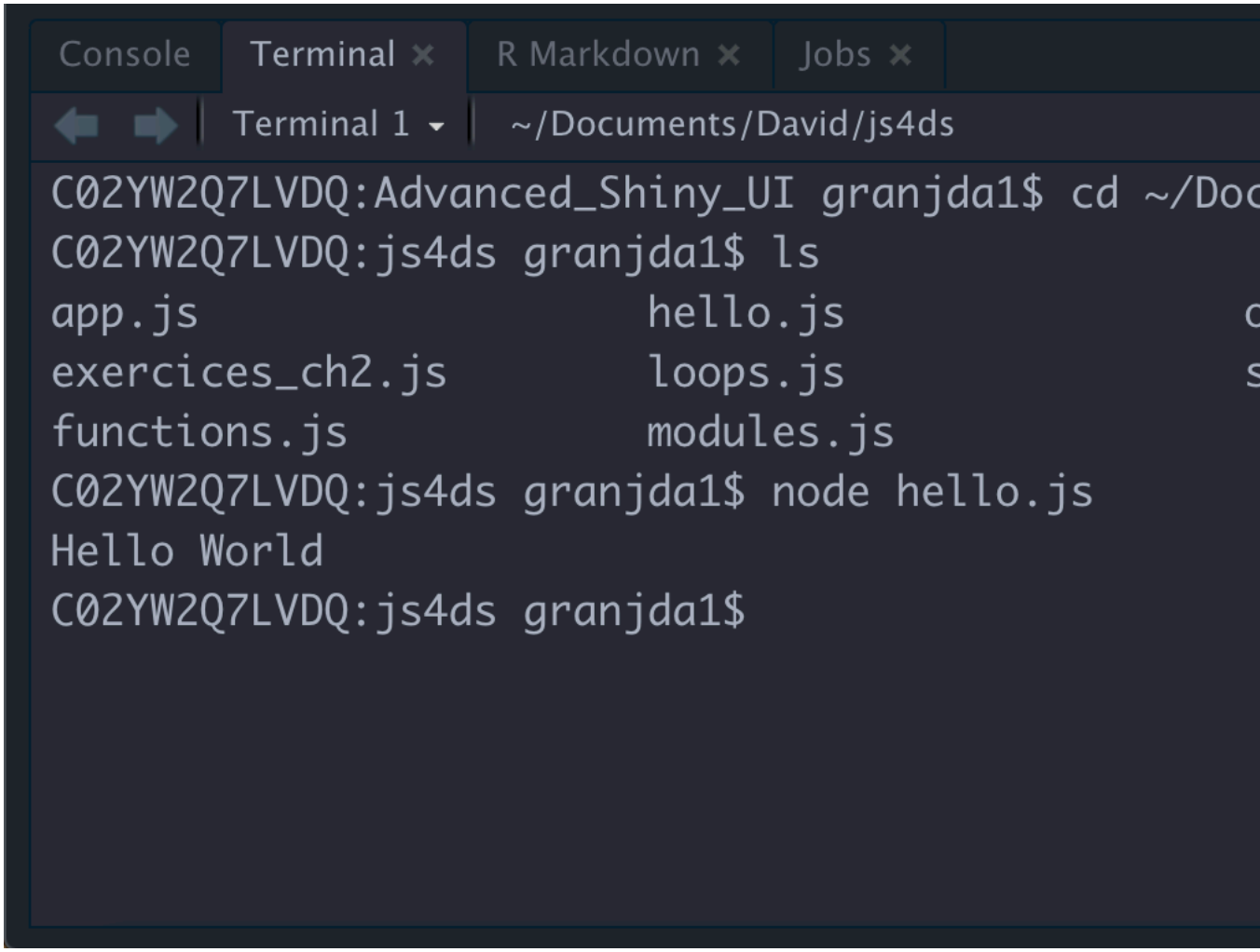


Figure 3.3: Run JS in a terminal

3.3 Programming with JS: basis

We are now all set to introduce the basis of JS. As many languages, JS is made of variables and instructions. All instructions end by the `;` symbol.

3.3.1 JS types

JS defines several types:

- Number: does not distinguish between integers and others (in R for instance, numeric contains integers and double)
- String: characters ('blabla')
- Boolean: true/false

To check the type of an element, we may use the `typeof` operator.

```
typeof 1; // number
typeof 'pouic'; // string
```

In JS, `typeof` is not a function like in R!!! Therefore don't write `typeof('string');`.

3.3.2 Variables

Variables are key elements in programmation. They allow to store intermediate results and do other manipulations. In JS, a variable is defined by:

- a type
- a name
- a value

Valid variable names:

don't use an existing name like typeof

don't start with a number (123soleil)

don't include any space (total price)

Besides, code style is a critical element in programming, increasing readability, and general consistence. There are several styles, the main ones being `snake_case` and `camelCase`. I personally use the camelCase syntax to write variables in JS. To set a variable we use `let` (there exists `var` but this is not the latest JS norm (ESMAScript 6 or ES6). You will see later that we still use `var` in the shiny core and many other R packages).

There are two ways to create variables in JavaScript.

3.3.2.1 Const

In JavaScript, a variable may be created with `const`:

```
const n = 1;
n = 2; // error
const n = 3; // error
const a;
a = 1; // errors
```

As shown above, such variables:

- Cannot be modified
- Cannot share the same name
- Must be assigned a value

3.3.2.2 let

Another way to define a variable:

```
let myVariable = 'welcome';
myVariable = 1;
console.log(myVariable);
```

Then we may use all mathematical operators to manipulate our variables.

```
let myNumber = 1; // affectation
myNumber--; // decrement
console.log(myNumber); // print 0
```

List of numerical operators in JS:

+

-

*

/

% (modulo)

++ (incrementation)

– (decrementation)

To concatenate two strings, we use the+ symbol.

You may also know `var` to declare variables. What is the difference with `let`? It is mainly a scope reason:

```
var i = 1;
{
  var i = 2; // this will modify i globally, not locally
}
console.log(`i is ${i}`); // i is 2.

let j = 1;
{
  let j = 2; // j is only declared locally and not globally!
}
console.log(`j is ${j}`); // j is 1
```

3.3.3 Conditions

Below are the operators to check conditions.

== (A equal B)

!= (A not equal to B)

> (>=)

< (<=)

AND (A AND B)

OR (A OR B)

To test conditions there exists several ways:

- `if (condition) { console.log('Test passed'); }`
- `if (condition) { instruction A} else { instruction B }`

This is very common to other languages (and R for instance). Whenever a lot of possible conditions need to be evaluated, it is better to choose the `switch`.

```
switch (variable) {
  case val1: // instruction 1
  break; // don't forget the break!
  case val2:  // instruction 2
  break;
  default: // when none of val1 and val2 are satisfied
}
```

3.3.4 Objects

JavaScript is an object oriented programming language (like Python). An object is defined by:

- a type
- some properties
- some methods (to manipulate properties)

Let's define our first object below:

```
const me = {
  name : 'Divad',
  age : 29,
  music : '',
  printName: function() {
    console.log(`I am ${this.name}`);
  }
}

me.geek = true; // works (see const variables above)
console.log(JSON.stringify(me)); // print a human readable object.

console.log(me.name);
console.log(me.age);
console.log(me.music);
// don't repeat yourself!!!
for (let key in me) { // here is it ok to use `in`
 console.log(`me[${key}] is ${me[key]}`);
}

me.printName();

me = {
  name: 'Paul',
  age: 40
} // error (see const variables above)
```

Some comments on the above code:

- to access an object propertie, we use `object.propertie`
- to print a human readable version of the object `JSON.stringify` will do the job
- we introduced string interpolation with `${*}`. `*` may be any valid expression.

- methods are accessed like properties (we may also pass parameters). We use `this` to refer to the object itself. Take note, we will see it a lot!

In JavaScript, we can find already predefined objects to interact with arrays, dates.

3.3.4.1 Arrays

An array is a structure allowing to store informations for instance

```
const table = [1, 'plop'];
table.push('hello');
table = [2]; // error (as explain in above in the variable part)
console.log(table);
```

Array may be nested

```
const nested = [1, ['a', [1, 2, 3]], 'plop'];
console.log(nested);
```

In arrays, elements may be accessed by their index, but as mentionned before, the first index is 0 (not 1 like in R). A convenient way to print all arrays's elements is to use an iteration:

```
const nested = [1, ['a', [1, 2, 3]], 'plop'];
for (let i of nested) {
  console.log(i);
}

// or with the classic approach
for (let i = 0; i < nested.length; i++) {
  console.log(nested[i]);
}
```

Note that the `length` method returns the size of an array and is very convenient in for loops. Below is a table referencing the principal methods for arrays (we will use some of them later)

Method/Property	Description
length	Return the number of elements in an array
Join(string separator)	Transform an array in a string
concat(array1, array2)	Assemble 2 arrays
pop()	Remove the last element of an array

Method/Property	Description
shift()	Remove the first element of an array
unshift(el1, el2, ...)	Insert elements at the beginning of an array
push(el1, el2, ...)	Add extra elements at the end of an array
sort()	Sort array elements by increasing value of alphabetical order
reverse()	Symetric of sort()

Quite honestly, we mainly use `push` and `length` in the next chapters.

3.3.4.2 Strings

Below are the main methods related to the String object (character in R)

Method/Property/Operator	Description
+ (operator)	String concatenation
length	String length
indexOf()	Gives the position of the character following the input string
toLowerCase()	Put the string in small letters
toUpperCase()	Put the string in capital letters

3.3.4.3 Math

Below we mention some useful methods to handle mathematical objects

Method	Description
parseInt()	Convert a string to integer
parseFloat()	Conversion to floating number

All classic functions like `sqrt`, trigonometric functions are of course available. We call them with the `Math.*` prefix.

3.3.5 Iterations

Iterations allow to repeat an instruction or a set of instructions multiple times. Let's assume we have an array containing 100000 random numbers. How would you do to automatically print them? This a what we are going to see below!

3.3.5.1 For loops

The for loop has multiple uses. Below is a classic case where we start by defining the index (variable). We then set an upper bound (the array length) and we finish by incrementing the index value. The code between curly braces is then executed.

```
const table = [...Array(100).keys()]; // create an empty array of length 100 (so from
for (let i = 0; i < table.length; i++) {
    console.log(table[i]);
}
```

The way we created the array is a bit special and deserves some explanations:

- `Array` is a method to define a new array. We call it this way `Array(arrayLength)` since we don't want to write 100 values 1 by 1. But if you try `console.log(Array(10));` you will get `[ <10 empty items> ]`, meaning that 10 slots are available but nothing is inside yet.
- `keys` defines keys for each table index. As a reminder, since `Array(10)` is an object (check with `console.log(typeof Array(10));`) we may use `Array(10).keys()`. This creates an Array Iterator
- `...` is a spread syntax, and is called with an iterable object (see above)

NOTE: Contrary to R, JavaScript index starts from 0 (not from 1)! This is good to keep in mind when we will mix both R and JS.

Let's have a look at the `forEach` method for arrays (introduced in ES5):

```
const letters = ["a", "b", "c", "d"];
letters.forEach((letter) => {
  console.log(letter);
});
```

Below is another way to create a for loop (introduced in ES6):

```
const samples = ['blabla', 1, null]; // this is an array!
for (let sample of samples) {
 console.log(sample);
}
```

What loop `for` loop should we use? The answer is: it depends on the situation! Actually, there even exists other ways (replace `of` by `in` and you get the indexes of the array, like with the first code, but this is really not recommended).

3.3.5.2 Other iterations: while

While loops are another way to iterate, as long as the condition defined is TRUE. The incrementation step is done at the end of the instruction.

```
const h = 3; i = 0;
while (i <= h) {
  console.log(i);
  i++; // we need to increment to avoid infinite loop
}
```

3.3.6 Functions

Functions are useful to wrap a succession of instructions to accomplish a given task. Defining functions allows programmers to save time (less copy and paste, less search and replace), make less errors and easily share code. In modern JavaScript (ES6), functions are defined as follows:

```
const a = 1;
const fun = (parm1, parm2) => {
  console.log(a);
  let p = 3;
  return Math.max(parm1, parm2); // I use the Math object that contains the max method
}
let res = fun(1, 2);
console.log(res); // prints a and 2. a global
console.log(p); // fails because p was defined inside the function
```

This above functions computes the maximum of 2 provided numbers. Some comments about scoping rules: variables defined inside the function are available for the function, but are not available outside the function definition. It should be noted that functions may use global variables defined outside of it.

3.3.6.1 Export functions: about modules

What happens if you wrote 100 functions that you want to reuse in different scripts? To prevent copying and pasting, we will now introduce the concept of modules. Let's save the below function in a script `utils.js`:

```
const findMax = (parm1, parm2) => {
  return Math.max(parm1, parm2); // I use the Math object that contains the max method
}
```

```
module.exports = {
  findMax = findMax
}
```

Let's create a `test.js` script in the same folder that uses the `findMax` function. To do this, we need to import the corresponding module:

```
const {findMax} = require('./utils.js');
findMax(1, 2); // prints 2
```

In the next chapters, we will see that some of the underlying JS code to build custom shiny inputs share the same utils functions. Therefore, introducing modules is necessary.

3.3.7 Event listeners

When you explore a web application, clicking on a button usually triggers something like a computation, a modal or an alert. How does this work? In JavaScript, interactivity plays a critical role. Indeed, you want the web application to react to user inputs like mouse clicks, keyboard events. Below we introduce DOM events.

Let's consider a basic HTML button.

```
<button id="mybutton">Go!</button>
```

On the JavaScript side, we first capture the button element using its id selector (`getElementById`).

```
const btn = document.getElementById('mybutton');
```

We then apply the `addEventListener` method. In short, an event listener is a program that triggers when a given event occurs (we can add multiple event listeners per HTML element). It takes 2 main parameters:

- the event: click, change, mouseover, ...
- the function to call

```
btn.addEventListener('click', function() {
  alert('Thanks!');
});
```

We could compare the JavaScript events to Shiny observeEvent in which we are listenning to a specific user input.

3.4 jQuery

3.4.1 Introduction

jQuery is a famous JavaScript library providing a user friendly interface to manipulate the DOM and is present in almost all actual websites. It is slightly easier (understand more convenient to use) than vanilla JS, even though web developers tend to avoid it to go back to vanilla JS (Bootstrap 5, the next iteration of Bootstrap will not rely on jQuery anymore). To use jQuery in a webpage, we must include its code either by dowloading the code and putting the minified JS file in our HTML or setting a link to a CDN.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Including jQuery</title>
  <!-- How to include jQuery -->
  <script src="https://code.jquery.com/jquery-3.5.0.js"></script>
</head>
<body>

<p>Hello World</p>


<script>
$('p').css('color', 'red');
</script>

</body>
</html>
```

3.4.2 Syntax

Below is a minimal jQuery code representing its philosophy (“write less, do more.”):

```
$(selector).action();
```

The selector slot stands for any jQuery selector like class, id, element, [attribute], :input (will select all input elements) and many more. As a reminder, let’s consider the following example:

```
<p class="text">Hello World</p>
```

To select and interact with this element, we use JavaScript and jQuery:

```
let inner = document.getElementsByClassName('text').innerHTML; // vanilla JS
let inner = $('.text').html(); // jQuery
```

This is of course possible to chain selectors

```
<ul class="list">
  <li class="item">1</li>
  <li class="item">2</li>
  <li class="item">3</li>
  <li class="item" id="precious-item">4</li>
</ul>

<ul class="list" id="list2">
  <li class="item">1</li>
  <li class="item">2</li>
  <li class="item">3</li>
  <li class="item">4</li>
</ul>
```

```
let items = $('.list .item'); // will return an array containing 8 li tags
let otherItems = $('#list2 .item'); // will select only li tags from the second ul ele
let lists = $('ul'); // will return an array with 2 ul elements
let firstItem = $('#list2:first-child'); // will return the first li element of the se
```

jQuery is obviously simpler than pure JavaScript.

3.4.3 Useful functions

There exist filtering functions dedicated to simplify item selection. We gathered the one mostly used in Shiny below.

3.4.3.1 Travel in the DOM

Method	Description
children()	Get the children of each element passed in the selector (important: only travels a single level down the DOM tree)
first()	Given an list of elements, select the first item
last()	Given an list of elements, select the last item
find()	Look for a descendant of the selected element(s) that could be multiple levels down in the DOM
closest()	Returns the first ancestor matching the condition (travels up in the DOM)
filter()	Fine tune element selection by applying a filter. Only return element for which the condition is true
siblings()	Get all siblings of the selected element(s)
next()	Get the immediately following sibling
prev()	Get the immediately preceding sibling
not()	Given an existing set of selected elements, remove element(s) that match the given condition

3.4.3.2 Manipulate tags

Below is a list of the main jQuery methods to manipulate tags (adding class, css property…)

Method	Description
addClass()	Add class or multiple classes to the set of matched elements
hasClass()	Check if the matched element(s) have a given class
removeClass()	Remove class or multiple classes to the set of matched elements
attr()	Get or set the value of a specific attribute
after()	Insert content after
before ()	Insert content before
css()	Get or set a css property
remove()	Remove element(s) from the DOM
val()	Get the current value of the matched element(s)

TO DO: add more methods

3.4.4 Chaining jQuery methods

A lot of jQuery methods may be chained, that is like pipe operations in R.

```
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
  <li>Item 4</li>
  <li>Item 5</li>
</ul>
```

We end the chain by ; and each step is indent by 2 spaces in the right direction.

```
$('ul')
  .first()
  .css('color', 'green') // add some style with css
  .attr('id', 'myAwesomeItem') // add an id attribute
  .addClass('amazing-ul');
```

3.4.5 Iterations

Like in vanilla JavaScript, it is possible to do iterations in jQuery. Let's consider the following HTML elements.

```
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
```

We apply the `each` method to change the style of each matched element step by step.

```
$('li').each(function() {
  $(this).css('visibility', 'hidden'); // will hide all li items
});
```

The `map` methods has a different purpose. It creates a new object based on the provided one.

```
const items = [0, 1, 2, 3, 4, 5];
const threshold = 3;

let filteredItems = $.map(items, function(i) {
  // removes all items > threshold
  if (i > threshold)
    return null;
  return i;
});
```

3.4.6 Good practice

It is recommended to wrap any jQuery code as follows:

```
$(document).ready(function(){
  // your code
});

// or a shortcut

$(function() {
  // your code
});
```

Indeed, do you guess what would happen if you try to modify an element that does not even exist? The code above will make sure that the document is ready before starting any jQuery manipulation.

3.4.7 Events

In jQuery there exists a significant number methods related to events. Below are the most popular:

```
$(element).click(); // click event
$(element).change(); // trigger change on an element
$(element).on('click', function() {
 // whatever
}); // attach an event handler function. Here we add click for the example
$(element).one('click', function() {
 // whatever
}); // the difference with on is that one will trigger only once
$(element).resize(); // useful to trigger plot resize in Shiny so that they correctly fit their c
$(element).trigger('change') // similar to $(element).change(); You will find it in the Shiny co
```

The `.on` event is frequently used in Shiny since it allows to pass custom events which are not part of the JS prefined events. For instance shinydashboard relies on a specific HTML/JavaScript/CSS template including a homemade API for handling the dashboard events. Don't worry if this section is not clear at the moment. We will see practical examples in the following chapters.

3.4.8 Extending objects

A last feature we need to mention about jQuery is the ability to extend objects with additional properties and/or method.

```
// jQuery way
$(function() {
  let object1 = {
    apple: 0
  };
  $.extend(object1, {
    print: function() {
      console.log(this);
    }
  });
  object1.print();
});
```

With vanilla JS we would use `Object.defineProperty`:

```
// pure JavaScript
Object.defineProperty(object1, 'print', {
  value: function() {
    console.log(this);
  },
  writable: false
});
```

Chapter 4

Shiny: What's under the Hood?

In the 2 previous chapters, we quickly introduced HTML and JavaScript. In this chapter, we are going to see what Shiny has under the hood. Therefore, as mentionned in the book prerequisites, you should be quite familiar with Shiny if you want to get the most out of this chapter.

We will answer to the following questions:

- What web dependencies is Shiny based on?
- How is R/JavaScript communication achieved?
- How does Shiny deal with inputs?

4.1 Shiny, HTML, CSS and JavaScript

Shiny allows to develop web applications with R in minutes. Let's face it: this is quite mind blowing! Well, this won't probably be a production app but still a working prototype. Believe me, doing a web application with pure HTML/CSS and JavaScript is more difficult, especially for a non web developer background.

Is Shiny less customizable than a classic web app? Not at all! Indeed, Shiny has its own engine to build HTML tags, through R, meaning that all HTML elements are available. You may also include any custom JavaScript or CSS code.

Do you remember about the first experiment of Chapter 2? We only did

```
library(shiny)
p("Hello World")
```

and noticed that the `p` function generates HTML. We will study in chapter 5 the tools to build/modify/delete these tags. The main difference between HTML tags and Shiny tags is the absence of closing tag for Shiny. For instance, in raw HTML, we expect `<p>` to be closed by `</p>`. In Shiny, we only call `p(...)`, where `...` may be attributes like class/id or children tags.

4.2 Discover Shiny dependencies

The simplest Shiny layout is the `fluidPage`. The `shinyapp` predefined Rstudio snippet will create a basic app skeleton (type shinyapp in RStudio IDE):

```
ui <- fluidPage(
  p("Hello World")
)

server <- function(input, output, session) {}
shinyApp(ui, server)
```

At first glance, the page only contains text. Waiiit … are you sure about this? Let's run the above example and open the HTML inspector introduced in 2. Results are displayed on Figure 4.1.

We see in the head section that Shiny has 4 dependencies:

- json2
- jQuery 3.4.1
- shiny (custom JavaScript and CSS)
- Bootstrap 3.4.1 (JavaScript and CSS) + other files (html5shiv, respond)

Bootstrap is here to provide plug and play design and interactions (tabs, navs). For instance the `fluidRow` and `column` functions of Shiny leverage the Bootstrap grid to control how elements are displayed in a page. This is convenient because it avoids to write a crazy amount of CSS/JavaScript and always reinvent the wheel.

jQuery drives the DOM manipulations. Shiny has its own JS and CSS files. Finally, json2 is a library to handle the JSON data format (JavaScript Object Notation). In the following chapters we will use it a lot, through the jsonlite package that allows to transform JSON objects in R objects and inversely.

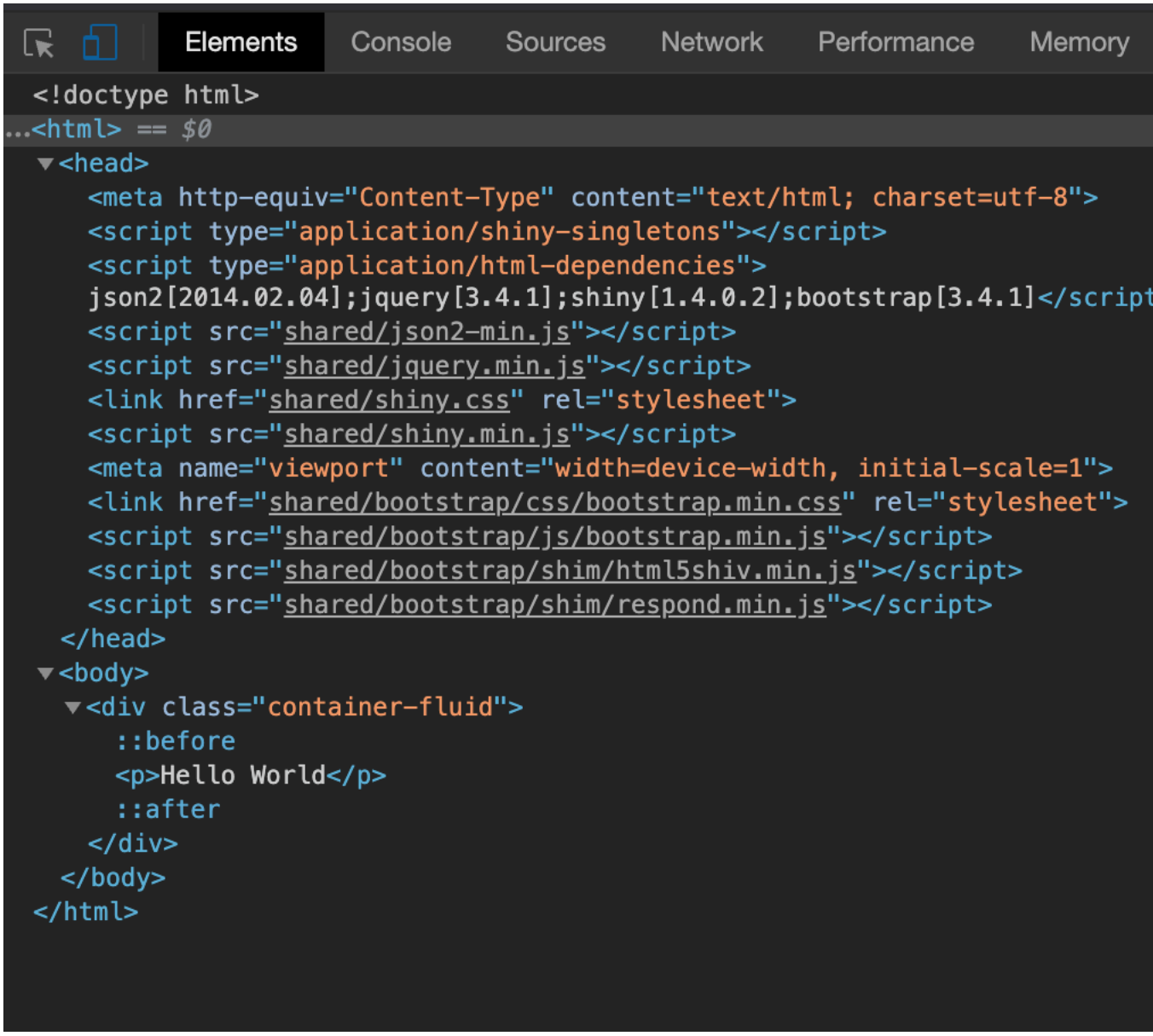


Figure 4.1: Shiny dependencies

In summary, all those libraries are necessary to make Shiny what it is! Customizing Shiny may imply to alter those existing libraries (except the Shiny core JavaScript and json2). In Chapter 6 we will discover better tools to extract HTML dependencies. Finally, in Chapter 12.4.1, we will see a special case to insert dependencies during the app runtime.

4.3 Websocket: R/JS bidirectional communication

How does R (server) and JavaScript (client) communicate? This is a builtin Shiny feature highlighted here, leveraging the httpuv and websocket packages. We will not detail how they work but rather how to inspect the websocket in a web browser. Let's run the following app.

```
shinyApp(
  ui = fluidPage(
    selectInput("variable", "Variable:",
                c("Cylinders" = "cyl",
                  "Transmission" = "am",
                  "Gears" = "gear")),
    tableOutput("data")
  ),
  server = function(input, output) {
    output$data <- renderTable({
      mtcars[, c("mpg", input$variable), drop = FALSE]
    }, rownames = TRUE)
  }
)
```

After opening the HTML inspector, we select the network tab and search for websocket in the list. We also choose the message tab to inspect what R and JavaScript say to each others. On the JavaScript side, the websocket is created in the shinyapp.js file. The first element received from R is the first message in the list shown in Figure 4.2. It is a JSON containing the method used as well as passed data. In the meantime, you may change the select input value.

```
socket.send(JSON.stringify({
  method: 'init',
  data: self.$initialInput
}));
```

The second message received from R is after updating the select input.

```
this.sendInput = function(values) {
  var msg = JSON.stringify({
    method: 'update',
    data: values
  });
// other things
};
```

All of this is quite complex but extremely useful to check wether input/output work properly. In case of error, we would see the field `error` containing some elements. In the last part of this book, we will be designing custom inputs and knowing how to debug them outside R is priceless.

Finally, `Shiny.shinyapp.$socket.readyState` returns the state of the socket connection. It should be 1 if your app is running but I've seen some cases where the socket was actually closed (and nothing could happen).

Understanding how R and JS may communicate will help in the next section about the Shiny input system. Note that the R option `options(shiny.trace = TRUE)` allows to display the websocket messages.

4.4 The Shiny session object

We won't be able to go anywhere without giving some reminders about the Shiny session object. Why do we say object? `session` is actually an instance of the `ShinySession` R6 class. The initialization takes one parameter, namely the websocket. As shown in the last section, the websocket allows bidirectional exchanges between R and JS:

- `sendCustomMessage` sends messages from R to JS. It calls the private `sendMessage` method, which itself calls `write`. The message is sent only when the session is opened, throught the websocket `private$websocket$send(json)`. If the `shiny.trace` option is TRUE, a message showing the sent JSON is displayed, useful for debugging.
- `sendInputMessage` is used to update inputs from the server

```
sendCustomMessage = function(type, message) {
  data <- list()
  data[[type]] <- message
  private$sendMessage(custom = data)
}

sendInputMessage = function(inputId, message) {
  data <- list(id = inputId, message = message)
```

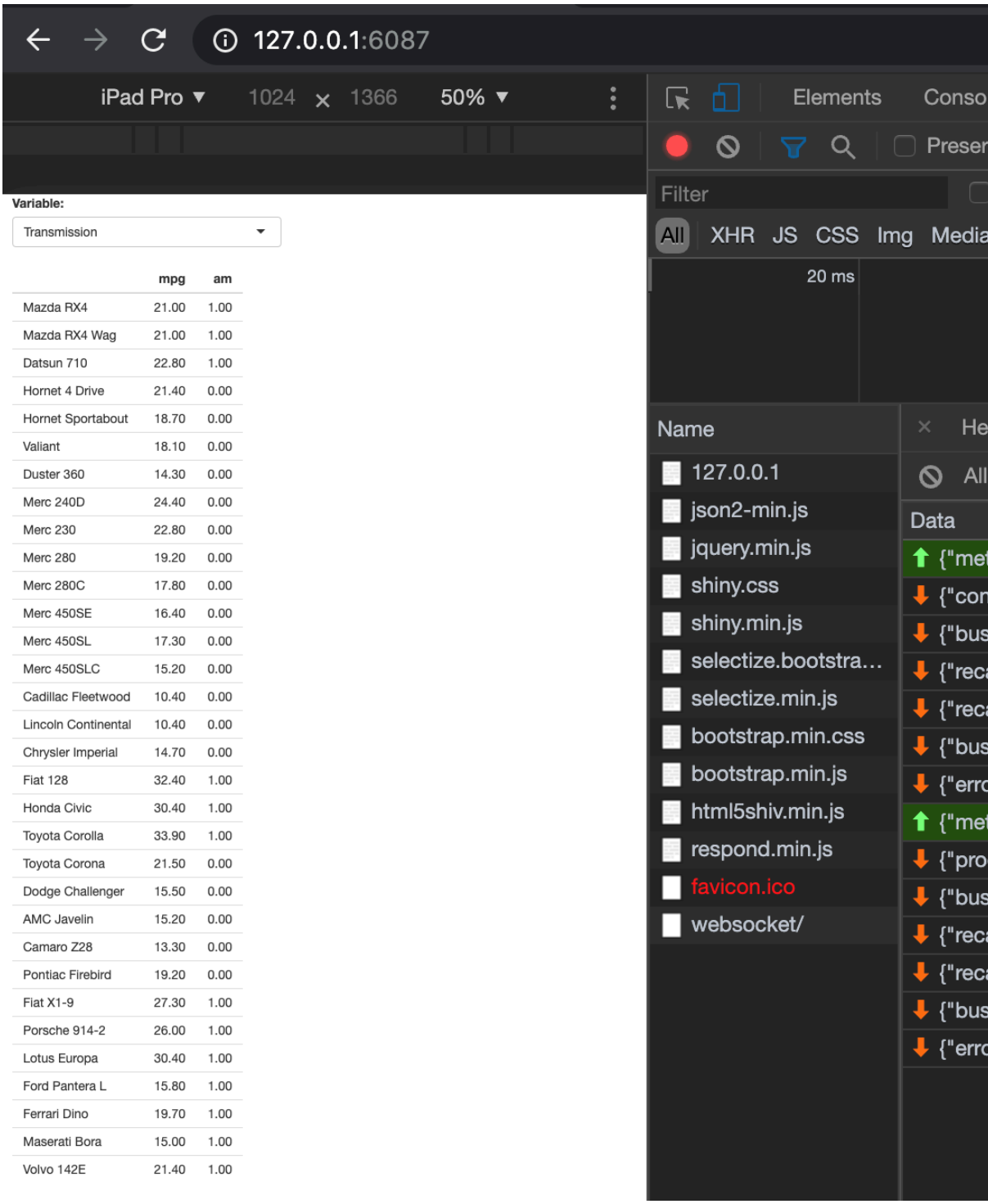


Figure 4.2: Shiny websocket

```
  # Add to input message queue
  private$inputMessageQueue[[length(private$inputMessageQueue) + 1]] <- data
  # Needed so that Shiny knows to actually flush the input message queue
  self$requestFlush()
}


sendMessage = function(...) {
  # This function is a wrapper for $write
  msg <- list(...)
  if (anyUnnamed(msg)) {
    stop("All arguments to sendMessage must be named.")
  }
  private$write(toJSON(msg))
}


write = function(json) {
  if (self$closed){
    return()
  }
  traceOption <- getOption('shiny.trace', FALSE)
  if (isTRUE(traceOption) || traceOption == "send")
    message('SEND ',
            gsub('(?m)base64,[a-zA-Z0-9+/=]+','[base64 data]',json,perl=TRUE))
  private$websocket$send(json)
}
# ...
```

No worry if it is not clear at the moment. We will discuss those elements in the following sections.

4.5 Shiny's input system

The goal of this part is to better understand how Shiny inputs work.

4.5.1 Shiny JavaScript sources

The Shiny input system relies on the Shiny JavaScript sources. They are located in the srcjs folder shown in Figure 4.3.

Notice the `_start.js` and `_end.js`. These will be used by the `Gruntfile.js`, that is a grunt-based tool to run different tasks such as concatenate multiple

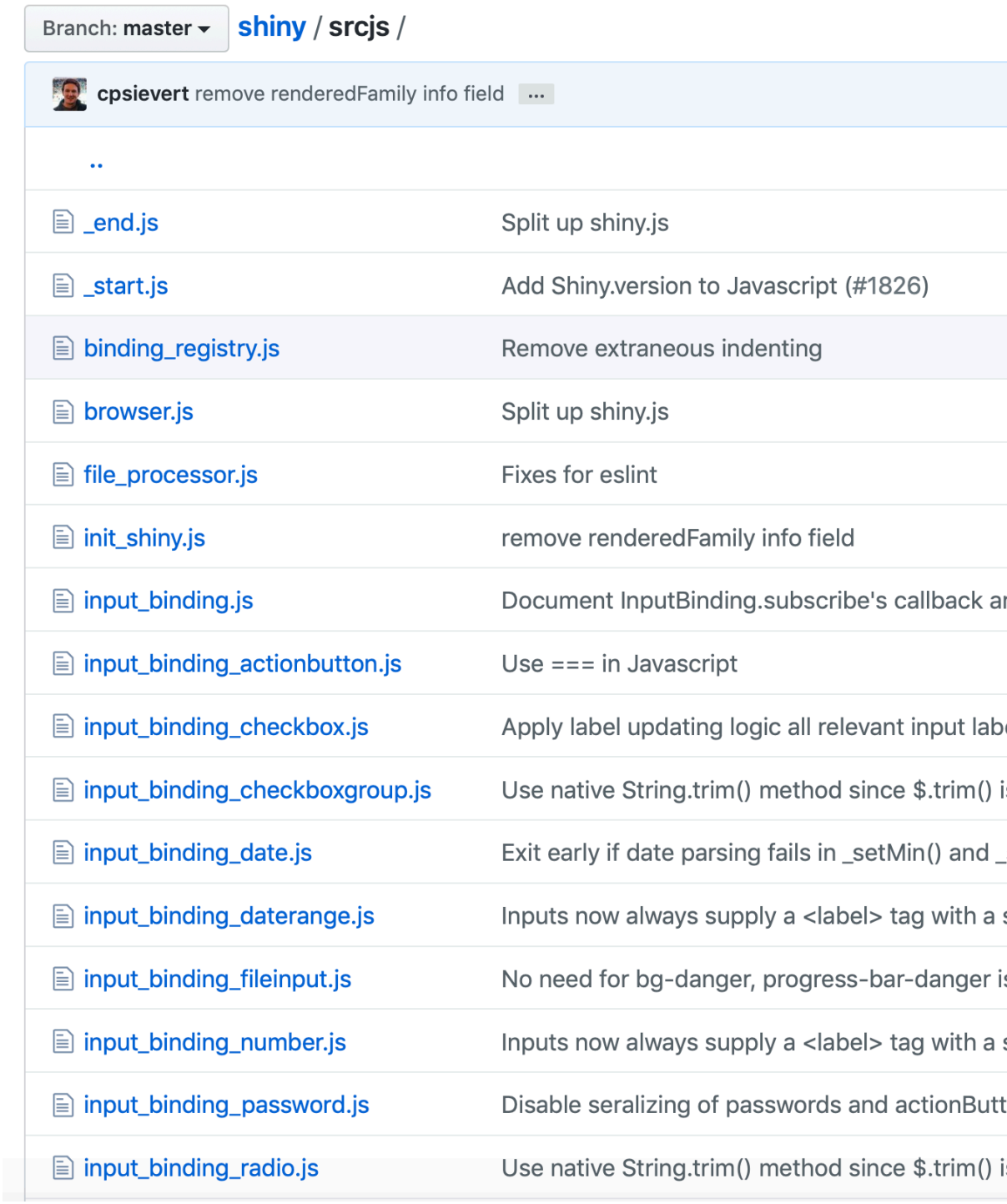


Figure 4.3: Shiny JavaScript sources

JavaScript files, lint the code, minify it … See here for a summary.

```
grunt.registerTask('default', [
'concat',
'string-replace',
'validateStringReplace',
'eslint',
'configureBabel',
'babel',
'uglify'
]);
```

These results in a large big file and its minified version in the `shiny/inst/www/shared` folder.

4.5.2 The Shiny JavaScript object

The `Shiny` object is exported at the top of the `shiny.js` file [1]. In other words, this means that we may use this object and any of its properties within the HTML inspector console tab, in any JavaScript file or shiny app as below.

```
ui <- fluidPage(
  tags$script(
    "$(function() {
      console.log(Shiny);
    });
    "
  )
)
server <- function(input, output, session) {}
shinyApp(ui, server)
```

This object contains many properties and methods as shown in Figure 4.4. We will discuss some of them later, like `Shiny.setInputValue`, `Shiny.addCustomMessageHandler`, `Shiny.shinyapps`, `Shiny.bindAll`, …

4.5.3 Initialization

When we run our app, most of the time it works and it is just fine! But what happens so that inputs and outputs are correctly handled? Upon initialization, Shiny runs several JavaScript functions. Not surprisingly, there is one called

[1] Refer to Chapter 3 if you don't remember how to export an object and make it available to all JS files.

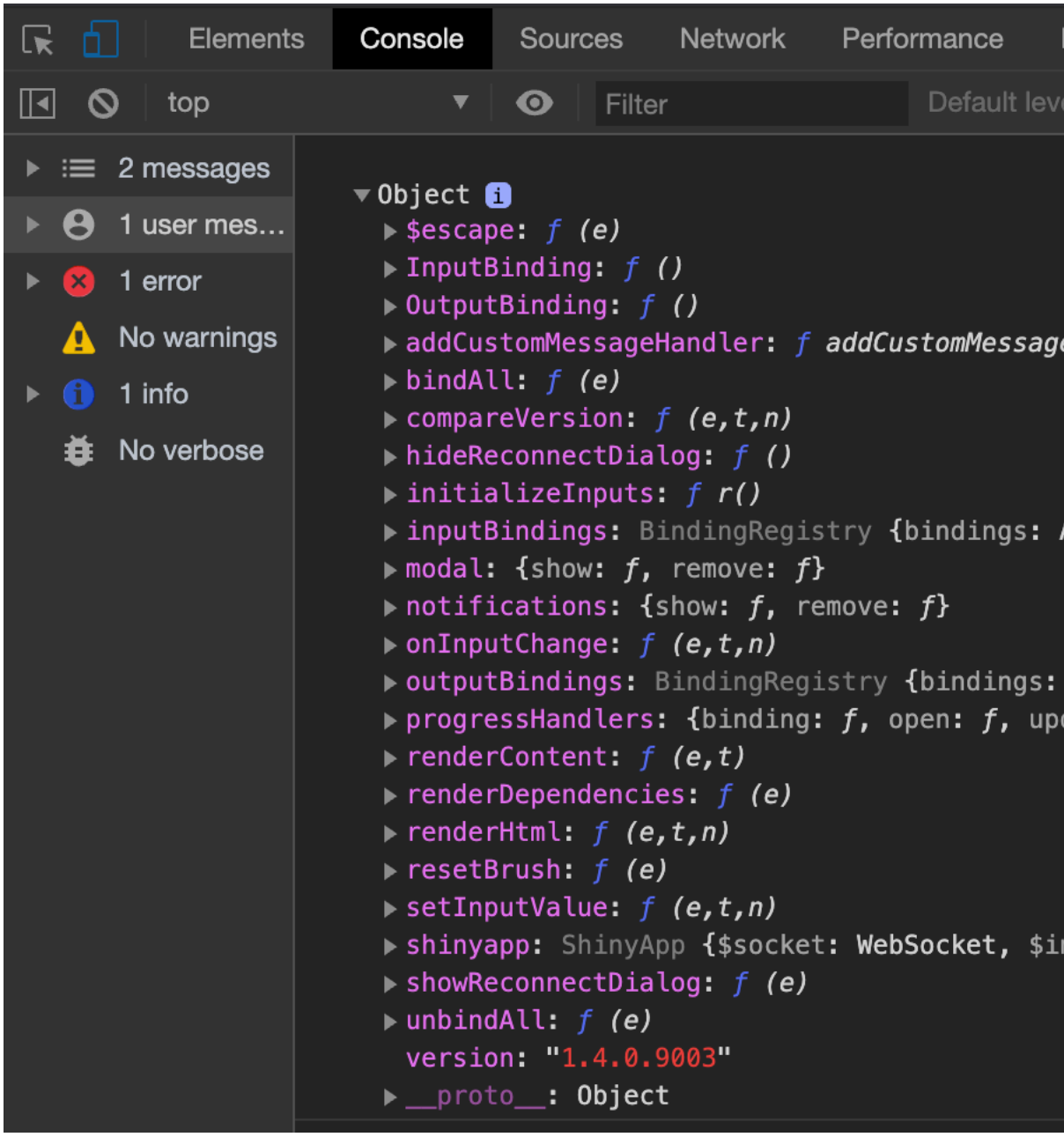


Figure 4.4: The Shiny JavaScript object

`init_shiny` containing a subtantial amount of elements. We find utils functions like `bindOutputs`, `unbindOutputs` to respectively bind/unbind outputs, `bindInputs` and `unbindInputs` for inputs. Only `bindAll` and `unbindAll` are available to the user (see a usecase here). To illustrate what they do, let's run the app below.

```
ui <- fluidPage(
  sliderInput("obs", "Number of observations:",
              min = 0, max = 1000, value = 500
  ),
  plotOutput("distPlot")
)

server <- function(input, output, session) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs))
  })
}
shinyApp(ui, server)
```

We then open the HTML inspector and run `Shiny.unbindAll(document)` (document is the scope, that is where to search). Try to change the slider input. What do you observe? Now let's type `Shiny.bindAll(document)` and update the slider value. What happens? Magic isn't it? This simply shows that when inputs are not bound, nothing happens so binding inputs is necessary.

Let's see below what is an input binding and how it works.

4.5.4 Input bindings

4.5.4.1 Reminders about Shiny inputs

Shiny contains a lot of inputs like `sliderInput`, `checkboxInput`, `radioButtons`, `numericInput`, ...Their role is to provide a way for the user to change parameters and update the app state.

```
numericInput("obs", "Observations:", 10, min = 1, max = 100)
```

All input functions have the same common `inputId` parameter, to guarantee the uniqueness of the given input.

```
ui <- fluidPage(
  textInput("text", "My Text 1"),
  hr(),
```

```
  uiOutput("mytext")
)

server <- function(input, output, session) {
  output$mytext <- renderText(input$text)
}

shinyApp(ui, server)
```

Now the questions is: how does Shiny recognizes inputs and drives their behavior?

4.5.4.2 Binding Shiny inputs

An input binding allows Shiny to identify each instance of a given input and what you may do with this input. For instance, a slider input must update whenever the range is dragged or when the left and right arrows of the keyboard are pressed. It relies on a class defined in the `input_binding.js` file.

Let's describe each method chronologically.

4.5.4.2.1 Find the input

The first step, is critical and consists in locating the input in the DOM. We could compare this to the receptor/ligand reaction. On the R side, we define an input, with a specific attribute that will serve as a receptor for the binding. For most of inputs, the `type` attribute will suit. Sometimes it may also be the class, like for the `actionButton`. On the JS side, we need a method that will indentify this receptor. Moreover, two different types of inputs (for instance `radioButton` and `selectInput`) cannot have the same receptor for conflict reasons, whereas two instances of the same input type can (if your app contains 10 sliders, they all share the same input binding!). The receptor identifier is provided by the `find` method of the `InputBinding` class. This method must be applied on a scope, that is the `document`. `find` accepts any valid jQuery selector. Figure 4.5 summarizes this important step.

Below, we are going to create a new binding for the `textInput`, with only 2 methods mentionned in the previous section, that is `find` and `getValue`. For that, we need to create a customized textInput, `customTextInput` so that it is not recognized by the classic Shiny text input binding. We add the input-text class and make our own input binding pointing to that specific class.

```
customTextInput <- function (inputId, label, value = "", width = NULL, placeholder = NU
  value <- restoreInput(id = inputId, default = value)
  div(
```

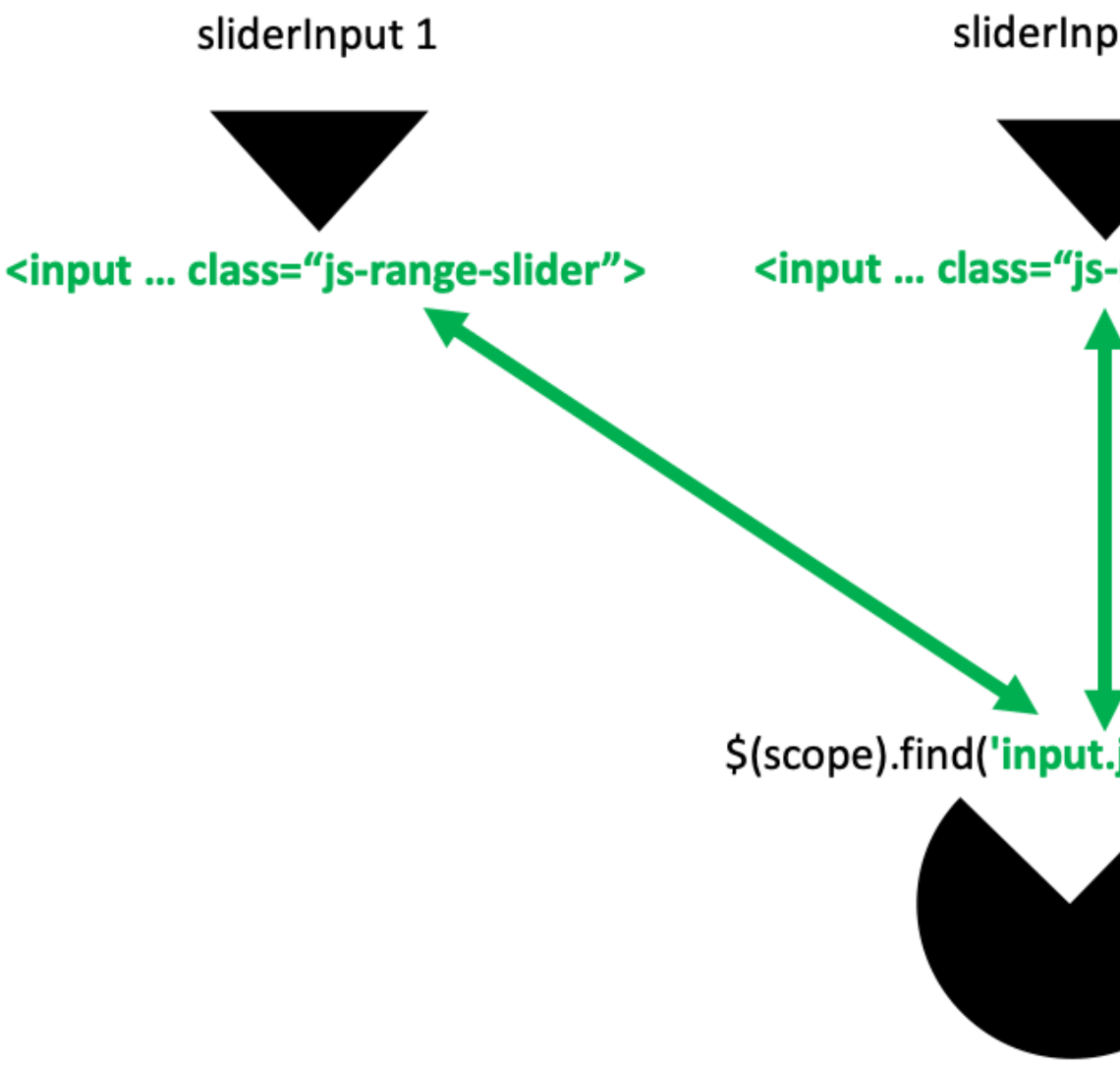


Figure 4.5: How to find inputs?

```
    class = "form-group shiny-input-container",
    style = if (!is.null(width)) {
      paste0("width: ", validateCssUnit(width), ";")
    },
    shiny:::shinyInputLabel(inputId, label),
    tags$input(
      id = inputId,
      type = "text",
      class = "form-control input-text",
      value = value,
      placeholder = placeholder
    )
  )
}
```

We put everything in a Shiny app and invit the reader to open the HTML inspector and look at the `console.log` result.

```
ui <- fluidPage(
  tags$script(
    "$(function() {
      // Input binding
      let customTextBinding = new Shiny.InputBinding();

      $.extend(customTextBinding, {
        find: function(scope) {
          console.log($(scope).find('.input-text'));
          return $(scope).find('.input-text');
        }
      });

      Shiny.inputBindings.register(customTextBinding, 'text');
    });
    "
  ),
  customTextInput("caption", "Caption", "Data Summary"),
  uiOutput("customText")
)

server <- function(input, output, session) {
  output$customText <- renderText(input$caption)
}
shinyApp(ui, server)
```

4.5.4.2.2 Initialize inputs

Upon initialization, Shiny calls the `initializeInputs` function that takes all input bindings and call their `initialize` method before binding all inputs. Note that once an input has been initialized it has a `_shiny_initialized` tag to avoid initializing it twice. The initialize method is not always defined but some API like Framework7, on top of which shinyMobile is built, require to have it. Below is an example for the toggle input:

```
// what is expected
let toggle = app.toggle.create({
  el: '.toggle',
  on: {
    change: function () {
      console.log('Toggle changed')
    }
  }
});
```

`app.toggle.create` is internal to the API. The corresponding shinyMobile input binding starts as follows.

```
var f7ToggleBinding = new Shiny.InputBinding();
  $.extend(f7ToggleBinding, {
    initialize: function(el) {
      app.toggle.create({el: el});
    },
    // other methods
});
```

Once initialized, we may use all specific methods provided by the API. Framework7 is clearly a gold mine, as its API provides a lot of possible options for many inputs/widgets.

4.5.4.2.3 Get the value

`getValue(el)` returns the input value. The way to obtain the value is different for almost all inputs. For instance, the `textInput` is pretty simple since the value is located in the `value` attribute. `el` refers to the element holding the id attribute and recognized by the `find` method. Figure 4.6 shows the result of a `console.log($(el));`.

To get the value, we apply the jQuery method `val` on the `$(el)` element and return the result.

```
ui <- fluidPage(
  tags$script(
```

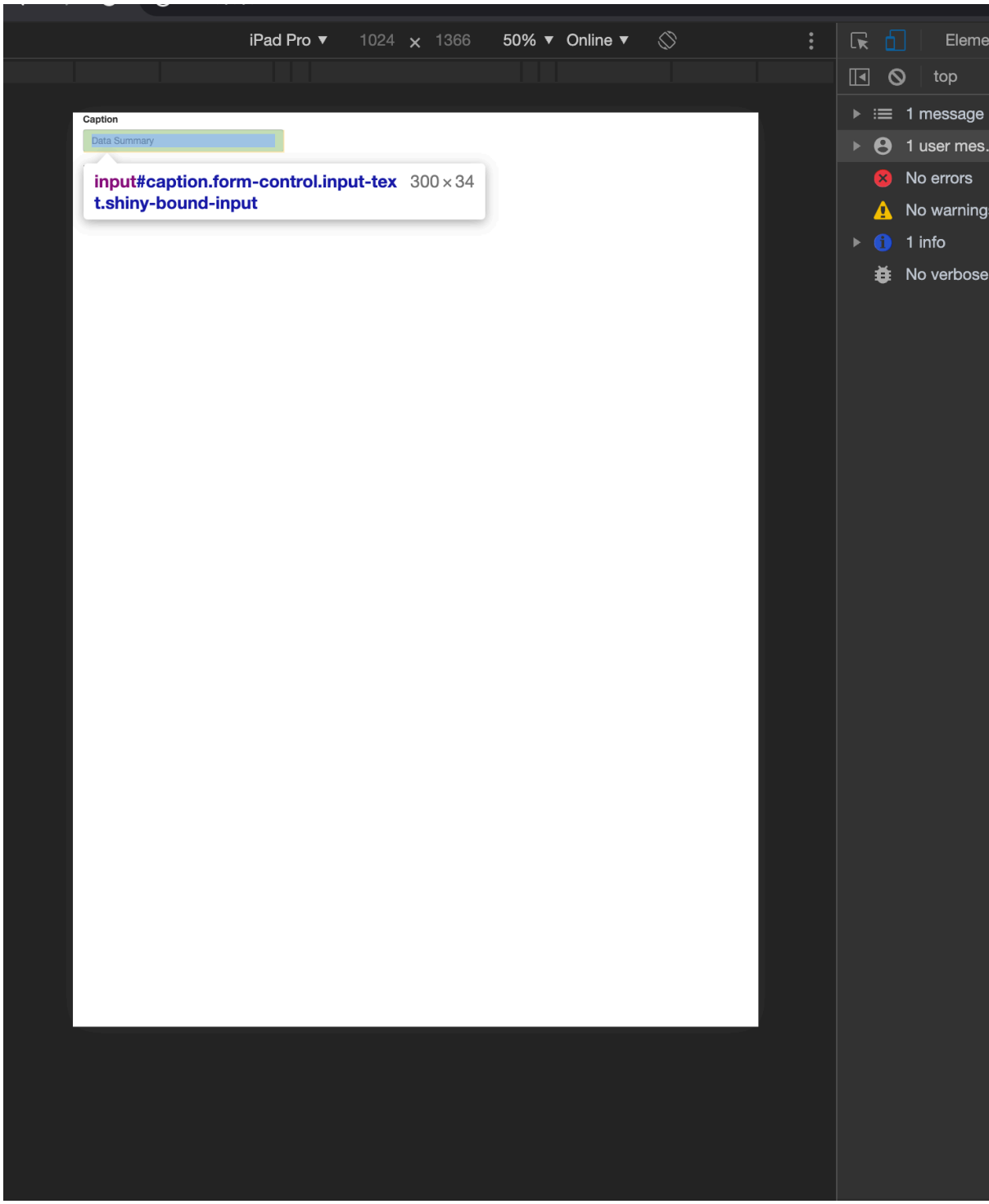


Figure 4.6: About el

```
    "$(function() {
      // Input binding
      let customTextBinding = new Shiny.InputBinding();

      $.extend(customTextBinding, {
        find: function(scope) {
          return $(scope).find('.input-text');
        },
        // Given the DOM element for the input, return the value
        getValue: function(el) {
          console.log($(el));
          return $(el).val();
        }
      });

      Shiny.inputBindings.register(customTextBinding, 'text');
    });
    "
  ),
  customTextInput("caption", "Caption", "Data Summary"),
  uiOutput("customText")
)

server <- function(input, output, session) {
  output$customText <- renderText(input$caption)
}
shinyApp(ui, server)
```

This time, the input value is returned. Notice that when you try to change the text content, the output value does not update as we would normally expect. We are actually missing a couple of methods so that the binding is fully working.

4.5.4.2.4 Set and update

`setValue(el, value)` is used to set the value of the current input. This method is necessary so that the input value can be updated. It has to be used in combination with `receiveMessage(el, data)`, which is the JavaScript part of all the R `updateInput` functions. We usually call the `setValue` method inside.

Let's create a function to update our custom text input. Call it `updateCustomTextInput`. It requires at least 3 parameters:

- inputId tells which input to update.
- value is the new value. This will be taken by the `setValue` JS method in the input binding

- session is the Shiny session object mentionned earlier. We will use the `sendInputMessage` to send values from R to JavaScript. The `receiveMessage` method will apply `setValue` with the data received from R

```
updateCustomTextInput <- function(inputId, value = NULL, session = getDefaultReactiveD
  session$sendInputMessage(inputId, message = value)
}
```

We add `setValue` and `receiveMessage` to custom input binding:

```
ui <- fluidPage(
  tags$script(
    "$(function() {
      // Input binding
      let customTextBinding = new Shiny.InputBinding();

      $.extend(customTextBinding, {
        find: function(scope) {
          return $(scope).find('.input-text');
        },
        // Given the DOM element for the input, return the value
        getValue: function(el) {
          return $(el).val();
        },
        setValue: function(el, value) {
          $(el).val(value);
        },
        receiveMessage: function(el, data) {
          console.log(data);
          this.setValue(el, data);
        }
      });

      Shiny.inputBindings.register(customTextBinding, 'text');
    });
    "
  ),
  customTextInput("caption", "Caption", "Data Summary"),
  actionButton("update", "Update text!", class = "btn-success"),
  uiOutput("customText")
)

server <- function(input, output, session) {
  output$customText <- renderText(input$caption)
```

```
  observeEvent(input$update, {
    updateCustomTextInput("caption", value = "new text")
  })
}
shinyApp(ui, server)
```

Figure 4.7 illustrates the main mechanisms.

If we have to pass multiple elements to update, we would have to change the `updateCustomTextInput` function such as:

```
updateCustomTextInput <- function(inputId, value = NULL, placeholder = NULL, session = getDefault
  message <- shiny:::dropNulls(
    list(
      value = value,
      placeholder = placeholder
    )
  )
  session$sendInputMessage(inputId, message)
}
```

`dropNulls` is an internal function ensuring that the list does not contain NULL elements. On the JS side, we would have to update the `receiveMessage` method. We send a list from R, which is then serialized to a JSON object. Properties like `value` may be accessed using the . notation:

```
receiveMessage: function(el, data) {
console.log(data);
if (data.hasOwnProperty('value')) {
this.setValue(el, data.value);
}

// other parameters to update...
}
```

So far so good! We managed to update the text input value. Yet, the output value does not change. We are going to fix this missing step in the next section.

4.5.4.2.5 Subscribe

`subscribe(el, callback)` listens to events telling under which circumstances to tell Shiny to update the input value and make it available in the app. Some API like Bootstrap explicitly mention those events (like `hide.bs.tab`, `shown.bs.tab`, …). Going back to our custom text input, what event would make it change?

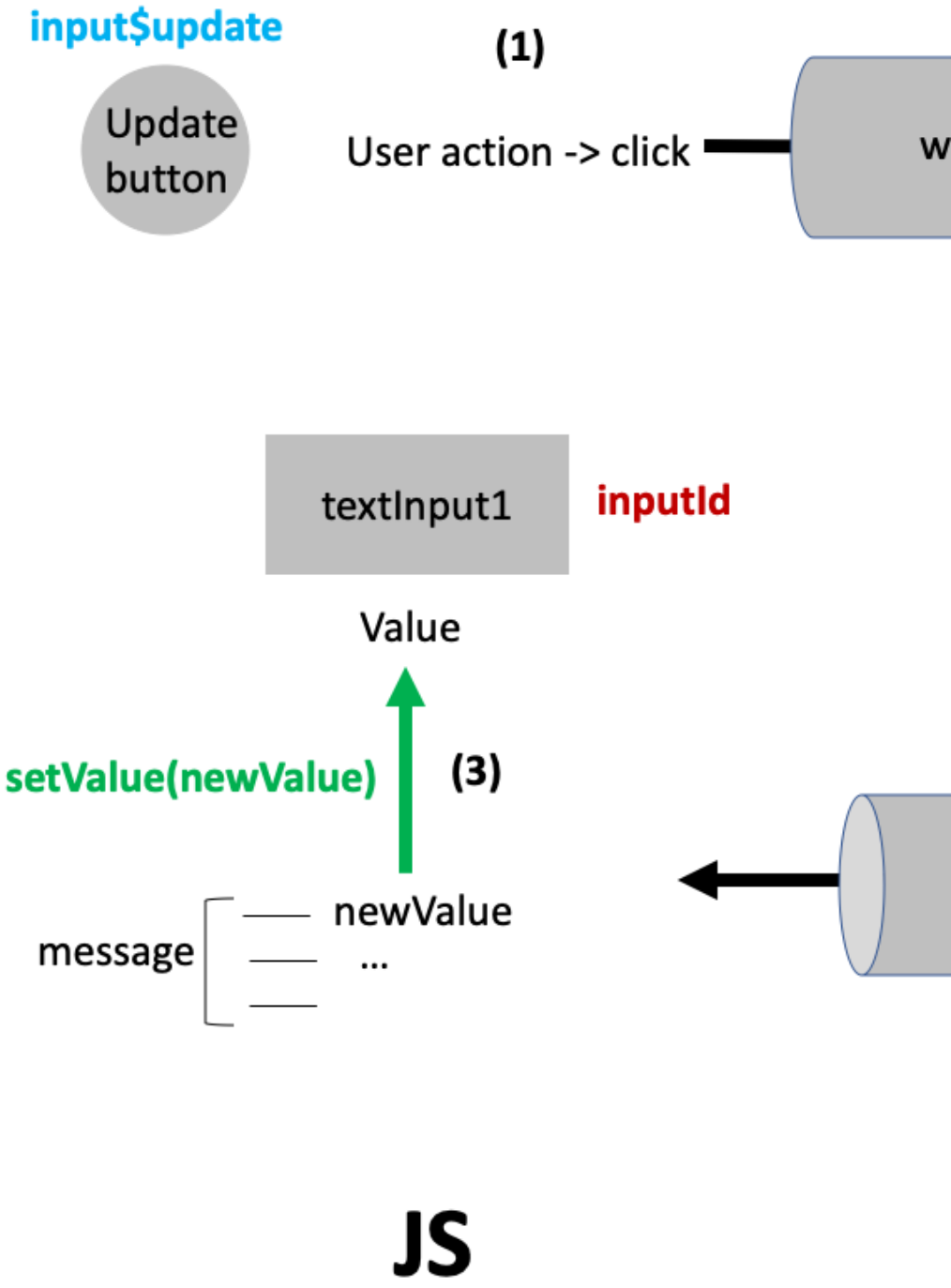


Figure 4.7: What happens after clickin on the update button?

- After a key is release on the keyboard. We may listen to `keyup`
- After copying and pasting any text in the input field or dictating text. The `input` event may be helpful

We may add those events to our binding using an event listener seen at the end of Chapter 3.

```
$(el).on('keyup.customTextBinding input.customTextBinding', function(event) {
callback(true);
});
```

`callback` ensures that the new value is captured by Shiny. We will come back later on the callback parameter.

```
ui <- fluidPage(
  tags$script(
    "$(function() {
      // Input binding
      let customTextBinding = new Shiny.InputBinding();

      $.extend(customTextBinding, {
        find: function(scope) {
          return $(scope).find('.input-text');
        },
        // Given the DOM element for the input, return the value
        getValue: function(el) {
          return $(el).val();
        },
        setValue: function(el, value) {
          $(el).val(value);
        },
        receiveMessage: function(el, data) {
          if (data.hasOwnProperty('value')) {
            this.setValue(el, data.value);
            $(el).trigger('change');
          }
        },
        subscribe: function(el, callback) {
          $(el).on('keyup.customTextBinding input.customTextBinding', function(event) {
            console.log(event);
            callback();
          });
        }
      });
```

```
      Shiny.inputBindings.register(customTextBinding, 'text');
    });
    "
  ),
  customTextInput("caption", "Caption", "Data Summary"),
  actionButton("update", "Update text!", class = "btn-success"),
  uiOutput("customText")
)

server <- function(input, output, session) {
  output$customText <- renderText(input$caption)
  observeEvent(input$update, {
    updateCustomTextInput("caption", value = "new text")
  })
}
shinyApp(ui, server)
```

Hurray! The output result is successfuly changed when the input value is manually changed. But nothing happens when we click on the update button. What did we miss? Looking back at the `receiveMessage` method, we change the input value but how does Shiny knows that this step was successful? To check that no event is raised, we put a `console.log(event);` in the `subscribe` method. Any action like removing the text content or adding new text triggers event but clicking on the action button does not. Therefore, we must trigger an event and add it to the `subscribe` method. We may choose the `change` event, that triggers when an element has finished changing.

```
$(el).on('change.customTextBinding', function(event) {
callback(false);
});
```

Let's try again.

```
ui <- fluidPage(
  tags$script(
    "$(function() {
      // Input binding
      let customTextBinding = new Shiny.InputBinding();

      $.extend(customTextBinding, {
        find: function(scope) {
          return $(scope).find('.input-text');
        },
        // Given the DOM element for the input, return the value
```

```
        getValue: function(el) {
          return $(el).val();
        },
        setValue: function(el, value) {
          $(el).val(value);
        },
        receiveMessage: function(el, data) {
          if (data.hasOwnProperty('value')) {
            this.setValue(el, data.value);
            $(el).trigger('change');
          }
        },
        subscribe: function(el, callback) {
          $(el).on('keyup.customTextBinding input.textInputBinding', function(event) {
            callback();
          });

          $(el).on('change.customTextBinding', function(event) {
            callback(false);
          });
        }
      });

      Shiny.inputBindings.register(customTextBinding, 'text');
    });
    "
  ),
  customTextInput("caption", "Caption", "Data Summary"),
  actionButton("update", "Update text!", class = "btn-success"),
  uiOutput("customText")
)

server <- function(input, output, session) {
  output$customText <- renderText(input$caption)
  observeEvent(input$update, {
    updateCustomTextInput("caption", value = "new text")
  })
}
shinyApp(ui, server)
```

Perfect? Not exactly. Wouldn't is be better to only change the input value once the keyboard is completly released for some time (and not each time a key is released). This is what we call debouncing. This implies to set a delay before telling Shiny to get the new value and the `getRatePolicy` method is exactly what we need. Additionally, we must also pass true to the `callback`

in the subscribe method, in order to apply our specific rate policy (debounce, throttle). This is useful for instance when we don't want to flood the server with useless update requests. For a slider, we only want to send the value as soon as the range stops moving and not all intermediate values. Those elements are defined here.

Run the app below and try to manually change the text input value by adding a couple of letters as fast as you can. What do you notice? We see that output value only updates when we release the keyboard.

```
ui <- fluidPage(
  tags$script(
    "$(function() {
      // Input binding
      let customTextBinding = new Shiny.InputBinding();

      $.extend(customTextBinding, {
        find: function(scope) {
          return $(scope).find('.input-text');
        },
        // Given the DOM element for the input, return the value
        getValue: function(el) {
          return $(el).val();
        },
        setValue: function(el, value) {
          $(el).val(value);
        },
        receiveMessage: function(el, data) {
          if (data.hasOwnProperty('value')) {
            this.setValue(el, data.value);
            $(el).trigger('change');
          }
        },
        subscribe: function(el, callback) {
          $(el).on('keyup.customTextBinding input.textInputBinding', function(event) {
            callback(true);
          });

          $(el).on('change.customTextBinding', function(event) {
            callback();
          });
        },
        getRatePolicy: function() {
          return {
            policy: 'debounce',
            delay: 250
```

```
          };
        },
        unsubscribe: function(el) {
          $(el).off('.customTextBinding');
        }
      });

      Shiny.inputBindings.register(customTextBinding, 'text');
    });
    "
  ),
  customTextInput("caption", "Caption", "Data Summary"),
  actionButton("update", "Update text!", class = "btn-success"),
  uiOutput("customText")
)

server <- function(input, output, session) {
  output$customText <- renderText(input$caption)
  observeEvent(input$update, {
    updateCustomTextInput("caption", value = "new text")
  })
}
shinyApp(ui, server)
```

You may adjust the delay according to your needs, but a too high delay would not feel very natural.

4.5.4.2.6 Register an input binding

At the end of the input binding definition, we register it for Shiny.

```
let myBinding = new Shiny.inputBinding();
$.extend(myBinding, {
// methods go here
});

Shiny.inputBindings.register(myBinding, 'reference');
```

Although the Shiny documentation mentions a `Shiny.inputBindings.setPriority` method to handle conflicting bindings, it is better not to have to use it.

4.5.4.3 Binding other elements

The Shiny input binding system is too convenient to only use it for input elements. In shinydashboard, you may know the `box` function. Boxes are contain-

ers with a title, body and footer as well as optional elements. Interestingly we may collapse/uncollapse the box. It would be nice to capture the state of the box in an input, so as to trigger other actions as soon as this input changes. Since an input value is unique, we must add an inputId parameter to the box function:

```
"%OR%" <- function(a, b) if (!is.null(a)) a else b

box2 <- function (..., inputId = NULL, title = NULL, footer = NULL, status = NULL, soli
                  background = NULL, width = 6, height = NULL, collapsible = FALSE,
                  collapsed = FALSE)
{
  boxClass <- "box"
  if (solidHeader || !is.null(background)) {
    boxClass <- paste(boxClass, "box-solid")
  }
  if (!is.null(status)) {
    boxClass <- paste0(boxClass, " box-", status)
  }
  if (collapsible && collapsed) {
    boxClass <- paste(boxClass, "collapsed-box")
  }
  if (!is.null(background)) {
    boxClass <- paste0(boxClass, " bg-", background)
  }
  style <- NULL
  if (!is.null(height)) {
    style <- paste0("height: ", validateCssUnit(height))
  }
  titleTag <- NULL
  if (!is.null(title)) {
    titleTag <- h3(class = "box-title", title)
  }
  collapseTag <- NULL
  if (collapsible) {
    buttonStatus <- status %OR% "default"
    collapseIcon <- if (collapsed)
      "plus"
    else "minus"
    collapseTag <- div(class = "box-tools pull-right", tags$button(class = paste0("btn
                                                                                   `data-widget` = "col
  }
  headerTag <- NULL
  if (!is.null(titleTag) || !is.null(collapseTag)) {
    headerTag <- div(class = "box-header", titleTag, collapseTag)
  }
```

```
  div(
    class = if (!is.null(width)) paste0("col-sm-", width),
    div(
      id = inputId,
      class = boxClass,
      style = if (!is.null(style))
        style,
      headerTag,
      div(class = "box-body", ...),
      if (!is.null(footer)) div(class = "box-footer", footer)
    )
  )
}
```

Besides, we create the `updateBox` function, which will collapse the box:

```
updateBox <- function(inputId, session = getDefaultReactiveDomain()) {
  session$sendInputMessage(inputId, message = NULL)
}
```

When collapsed, a box gets the `collapsed-box` class. This will be useful for the input binding. As mentionned above, it is also necessary to know when to tell Shiny to update the value with the `subscribe` method. Most of the time, the change event might suit but as shinydashboard is built on top of AdminLTE2, it has an API to control box behaviour. We identify 2 events corresponding to the collapsible action:

- expanded.boxwidget (Triggered after the box is expanded)
- collapsed.boxwidget (Triggered after the box is collapsed)

Those events are unfortunately not possible to use since the AdminLTE code forget to trigger them in the main JS code (see the collapse method line 577-612). There are other solutions, as shown below with the `click` event.

There is also a plug and play `toggleBox` method. To unleash the power of our box, we need to activate it with `$('#<box_id>').activateBox();` before the binding step. If you remember, the `initialize` method is exactly doing this:

```
let boxBinding = new Shiny.InputBinding();
$.extend(boxBinding, {
  initialize: function(el) {
    $(el).activateBox(); // box activation
  },
  find: function(scope) {
```

```
    return $(scope).find('.box');
  },
  getValue: function(el) {
    let isCollapsed = $(el).hasClass('collapsed-box')
    return {collapsed: isCollapsed}; // this will be a list in R
  },
  setValue: function(el, value) {
    $(el).toggleBox();
  },
  receiveMessage: function(el, data) {
    this.setValue(el, data);
    $(el).trigger('change');
  },
  subscribe: function(el, callback) {
    $(el).on('click', '[data-widget="collapse"]', function(event) {
      setTimeout(function() {
        callback();
      }, 550);
    });

    $(el).on('change', function(event) {
      setTimeout(function() {
        callback();
      }, 550);
    });
  },
  unsubscribe: function(el) {
    $(el).off('.boxBinding');
  }
});

Shiny.inputBindings.register(boxBinding, 'box-input');
```

Some comments about the binding:

- `getValue` returns an object which will give a list in R. This is in case we add other elements like the remove action available in AdminLTE
- `setValue` calls the plug and play `toggleBox` method
- `receiveMessage` must trigger a change event so that Shiny knows when the value needs to be updated
- `subscribe` listens to the `click` event on the `[data-widget="collapse"]` element and delays the `callback` call by a value which is slightly higher than the default AdminLTE2 animation to collapse the box (500mx). If you omit this part, the input will not have time to properly update!!!

- We don't need extra listener for the `updateBox` function since it also triggers a click on the collapse button, thereby forwarding to the corresponding listener

Let's try our new toy in a simple dashboard:

```
shinyApp(
  ui = dashboardPage(
    dashboardHeader(),
    dashboardSidebar(),
    dashboardBody(
      tags$script(
        "$(function() {
          let boxBinding = new Shiny.InputBinding();
          $.extend(boxBinding, {
            initialize: function(el) {
              $(el).activateBox();
            },
            find: function(scope) {
              return $(scope).find('.box');
            },
            getValue: function(el) {
              let isCollapsed = $(el).hasClass('collapsed-box');
              return {collapsed: isCollapsed}; // this will be a list in R
            },
            setValue: function(el, value) {
              $(el).toggleBox();
            },
            receiveMessage: function(el, data) {
              this.setValue(el, data);
              $(el).trigger('change');
            },
            subscribe: function(el, callback) {
              $(el).on('click', '[data-widget=\"collapse\"]', function(event) {
                setTimeout(function() {
                  callback();
                }, 550);
              });
              $(el).on('change', function(event) {
                setTimeout(function() {
                  callback();
                }, 550);
              });
            },
            unsubscribe: function(el) {
```

```
              $(el).off('.boxBinding');
            }
          });

          Shiny.inputBindings.register(boxBinding, 'box-input');
        });
        "
      ),

      box2(
        title = textOutput("box_state"),
        "Box body",
        inputId = "mybox",
        collapsible = TRUE,
        plotOutput("plot")
      ),
      actionButton("toggle_box", "Toggle Box")
    ),
    title = "Dashboard example"
  ),
  server = function(input, output) {

    output$plot <- renderPlot({
      req(!input$mybox$collapsed)
      plot(rnorm(200))
    })

    output$box_state <- renderText({
      state <- if (input$mybox$collapsed) "collapsed" else "uncollapsed"
      paste("My box is", state)
    })

    observeEvent(input$toggle_box, {
      updateBox("mybox")
    })

  }
)
```

4.5.5 Utilities to quickly define new inputs

If you ever wondered where the `Shiny.onInputChange` or `Shiny.setInputValue` comes from (see article), it is actually defined in the `initShiny` function.

```
exports.setInputValue = exports.onInputChange = function(name, value, opts) {
opts = addDefaultInputOpts(opts);
inputs.setInput(name, value, opts);
};
```

Briefly, this function avoids to create an input binding. It is faster to code but there is a price to pay: you lose the possibility to easily update the new input. Indeed, all input functions like `sliderInput` have their own update function like `updateSliderInput`, because of the custom input binding system (We will see it very soon)!

4.5.6 Miscellaneous

We present here some tools that may be useful...TO FINISH

4.5.6.1 Get access to initial values

Something we may notice when exploring the `initShiny` function is the existence of a `Shiny.shinyapp` object, defined as follows:

```
var shinyapp = exports.shinyapp = new ShinyApp();
```

Let's explore what `shinyApp` contains. The definition is located in the shinyapps.js script.

```
var ShinyApp = function() {
this.$socket = null;

// Cached input values
this.$inputValues = {};

// Input values at initialization (and reconnect)
this.$initialInput = {};

// Output bindings
this.$bindings = {};

// Cached values/errors
this.$values = {};
this.$errors = {};

// Conditional bindings (show/hide element based on expression)
this.$conditionals = {};
```

```
this.$pendingMessages = [];
this.$activeRequests = {};
this.$nextRequestId = 0;

this.$allowReconnect = false;
};
```

It creates several properties, some of them are easy to guess like `inputValues` or `initialInput`. Let's run the example below and open the HTML inspector. Notice that the `sliderInput` is set to 500 at `t0` (initialization).

```
ui <- fluidPage(
  sliderInput("obs", "Number of observations:",
              min = 0, max = 1000, value = 500
  ),
  plotOutput("distPlot")
)

server <- function(input, output, session) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs))
  })
}
shinyApp(ui, server)
```

Figure 4.8 shows how to access Shiny's initial input value with `Shiny.shinyapp.$initialInput.obs`. After changing the slider position, its value is given by `Shiny.shinyapp.$inputValues.obs`. `$initialInput` and `$inputValues` contains way more elements but we are only interested by the slider in this example.

I acknowledge, the practical interest might be limited but still good to know for debugging purposes.

4.6 Custom handlers: from R to JavaScript

Shiny contains tools to ease the communication between R and JavaScript. This is what happens in the last part. If you remember, we were playing with a `selectInput` and a `datatable`. How does R send messages to JavaScript?

We already discussed the usage of `sendInputMessage()` in the input binding section. The other important method is `sendCustomMessage(type, message)`. It works by pair with the JS method `Shiny.AddCustomMessageHandler`, linked with the type parameter.

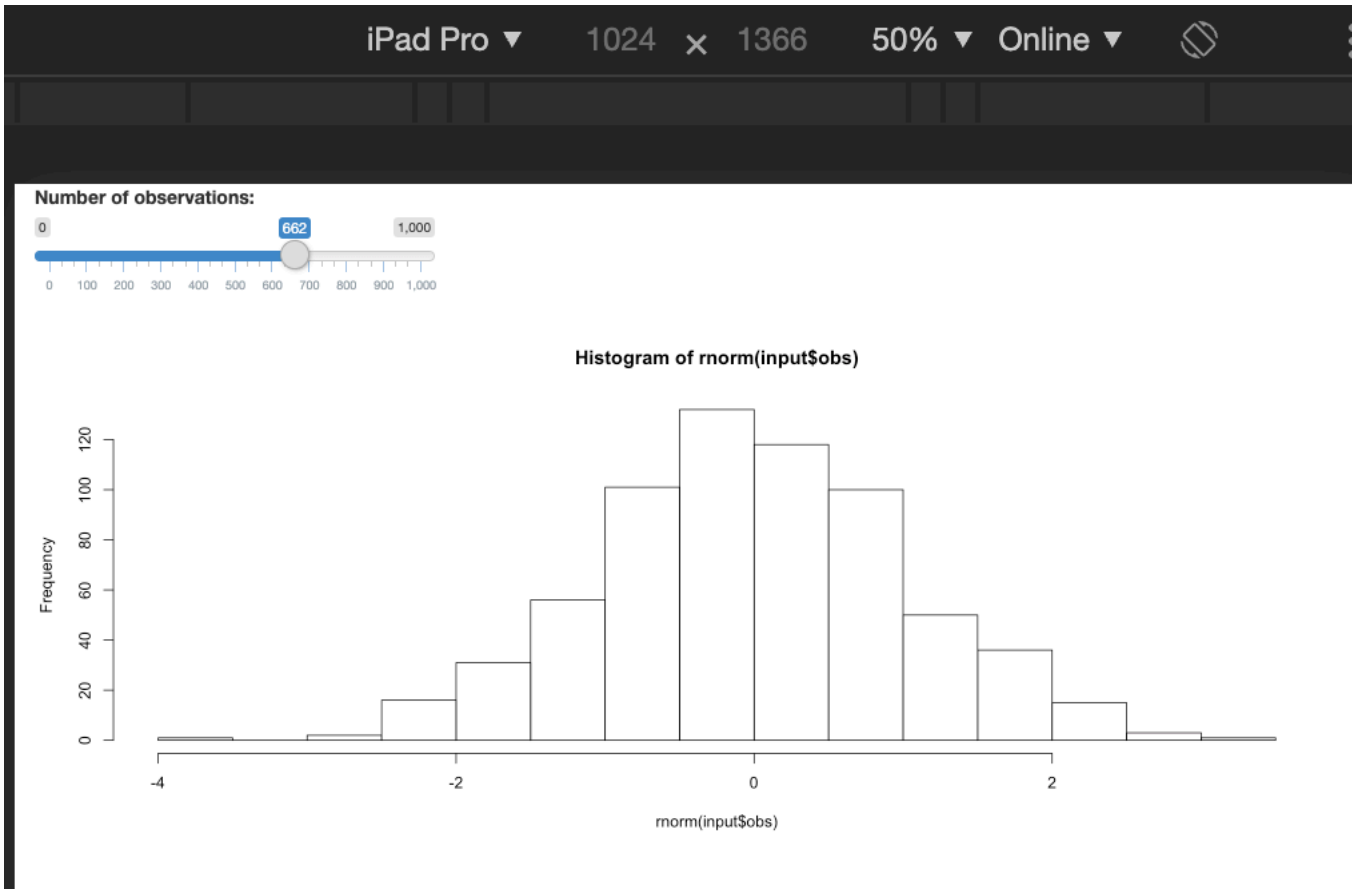


Figure 4.8: Explore initial input values

```
sayHelloToJS <- function(text, session = getDefaultReactiveDomain()) {
  session$sendCustomMessage(type = 'say-hello', message = text)
}
```

The JavaScript receptor is defined below:

```
$(function() {
  Shiny.AddCustomMessageHandler('say-hello', function(message) {
    alert(`R says ${message} to you!`)
  });
});
```

The shiny app below will simply print a welcome message. We obviously set `options(shiny.trace = TRUE)`.

```
options(shiny.trace = TRUE)
ui <- fluidPage(
  tags$head(
    tags$script(
      "$(function() {
        Shiny.addCustomMessageHandler('say-hello', function(message) {
          alert(`R says ${message} to you!`);
        });
      });
      "
    )
  )
)

server <- function(input, output, session) {
  observe({
    invalidateLater(5000)
    sayHelloToJS("hello")
  })
}

shinyApp(ui, server)
```

You will find a whole chapter dedicated to custom handlers here 12.

TO DO: picture showing the communication

htmltools

While building a custom html template, you will need to know more about the wonderful htmltools developed by Winston Chang, member of the shiny core team. It has the same spirit as devtools, that is, making your web developer life easier. What follows does not have the pretention to be an exhaustive guide about this package. Yet, it will provide you yith the main tools to be more efficient.

Chapter 5

htmltools overview

5.1 HTML Tags

htmltools contains tools to write HTML tags that were introduced in Chapter 2:

```
div()
```

If you had to gather multiple tags together, prefer `tagList()` as `list()`, although the HTML output is the same. The first has the shiny.tag.list class in addition to list. (The Golem package allows to test if an R object is a tag list. In this case, using a list would cause the test fail).

5.2 Notations

Whether to use `tags$div` or `div` depends if the tag is exported by default. For instance, you could use `htmltools::div` but not `htmltools::nav` since nav does not have a dedicated function (only for p, h1, h2, h3, h4, h5, h6, a, br, div, span, pre, code, img, strong, em, hr). Rather use `htmltools::tags$nav`. Alternatively, there exists a function (in shiny and htmltools) called `withTags()`. Wrapping your code in this function enables you to use `withTags(nav(), ...)` instead of `tags$nav()`, thereby omitting the `tags$` prefixes.

5.3 Adding new tags

The `tag` function allows to add extra HTML tags not already defined. You may use it as follows:

```
tag("test", list(class = "test", p("Custom Tag")))
# structure below
 tag
 "test"
  list
 class = "test"
  p
 "Custom Tag"
```

5.4 Alternative way to write tags

htmltools comes with the `HTML()` function that you can feed with raw HTML:

```
HTML('<div>Blabla</div>')
# will render exactly like
div("Blabla")

# but there class is different
class(HTML('<div>Blabla</div>'))
class(div("Blabla"))
```

You will not be able to use tag related functions, as in the following parts. Therefore, I strongly recommand using R and not mixing HTML in R. Interestingly, if you want to convert raw HTML to R code, there is a Shiny App developed by Alan Dipert from RStudio, namely html2R. There are some issues, non standard attributes (like `data-toggle`) are not correctly processed but there are solutions. This will save you precious time!

5.5 Playing with tags

5.5.1 Tags structure

According to the `tag` function, a tag has:

- a name such as span, div, h1 ... `tag$name`
- some attributes, which you can access with `tag$attribs`
- children, which you can access with `tag$children`
- a class, namely "shiny.tag"

For instance:

```
# create the tag
myTag <- div(
  class = "divclass",
  id = "first",
  h1("Here comes your baby"),
  span(class = "child", id = "baby", "Crying")
)
# access its name
myTag$name
# access its attributes (id and class)
myTag$attribs
# access children (returns a list of 2 elements)
myTag$children
# access its class
class(myTag)
```

How to modify the class of the second child, namely span?

```
second_children <- myTag$children[[2]]
second_children$attribs$class <- "adult"
myTag
# Hummm, this is not working ...
```

Why is this not working? By assigning `myTag$children[[2]]` to second_children, `second_children$attribs$class <- "adult"` modifies the class of the copy and not the original object. Thus we do:

```
myTag$children[[2]]$attribs$class <- "adult"
myTag
```

In the following section we explore helper functions, such as `tagAppendChild` from htmltools.

5.5.2 Useful functions for tags

htmltools and Shiny have powerful functions to easily add attributes to tags, check for existing attributes, get attributes and add other siblings to a list of tags.

5.5.2.1 Add attributes

- `tagAppendAttributes`: this function allow you to add a new attribute to the current tag.

For instance, assuming you created a div for which you forgot to add an id attribute:

```
mydiv <- div("Where is my brain")
mydiv <- tagAppendAttributes(mydiv, id = "here_it_is")
```

You can pass as many attributes as you want, including non standard attributes such as `data-toggle` (see Bootstrap 3 tabs for instance):

```
mydiv <- tagAppendAttributes(mydiv, `data-toggle` = "tabs")
# even though you could proceed as follows
mydiv$attribs[["aria-controls"]] <- "home"
```

5.5.2.2 Check if tag has specific attribute

- `tagHasAttribute`: to check if a tag has a specific attribute

```
# I want to know if div has a class
mydiv <- div(class = "myclass")
has_class <- tagHasAttribute(mydiv, "class")
has_class
# if you are familiar with %>%
has_class <- mydiv %>% tagHasAttribute("class")
has_class
```

5.5.2.3 Get all attributes

- `tagGetAttribute`: to get the value of the targeted attributes, if it exists, otherwise NULL.

```
mydiv <- div(class = "test")
# returns the class
tagGetAttribute(mydiv, "class")
# returns NULL
tagGetAttribute(mydiv, "id")
```

5.5.2.4 Set child/children

- `tagSetChildren` allows to create children for a given tag. For instance:

```
mydiv <- div(class = "parent", id = "mother", "Not the mama!!!")
# mydiv has 1 child "Not the mama!!!"
mydiv
children <- lapply(1:3, span)
mydiv <- tagSetChildren(mydiv, children)
# mydiv has 3 children, the first one was removed
mydiv
```

Notice that `tagSetChildren` removes all existing children. Below we see another set of functions to add children while conserving existing ones.

5.5.2.5 Add child or children

- `tagAppendChild` and `tagAppendChildren`: add other tags to an existing tag. Whereas `tagAppendChild` only takes one tag, you can pass a list of tags to `tagAppendChildren`.

```
mydiv <- div(class = "parent", id = "mother", "Not the mama!!!")
otherTag <- span("I am your child")
mydiv <- tagAppendChild(mydiv, otherTag)
```

You might wonder why there is no `tagRemoveChild` or `tagRemoveAttributes`. Let's look at the `tagAppendChild`

```
tagAppendChild <- function (tag, child) {
  tag$children[[length(tag$children) + 1]] <- child
  tag
}
```

Below we write the `tagRemoveChild`, where tag is the target and n is the position to remove in the list of children:

```
mydiv <- div(class = "parent", id = "mother", "Not the mama!!!", span("Hey!"))

# we create the tagRemoveChild function
tagRemoveChild <- function(tag, n) {
  # check if the list is empty
  if (length(tag$children) == 0) {
    stop(paste(tag$name, "does not have any children!"))
  }
  tag$children[n] <- NULL
  tag
}
```

```
mydiv <- tagRemoveChild(mydiv, 1)
mydiv
```

When defining the `tagRemoveChild`, we choose `[` instead of `[[` to allow to select multiple list elements:

```
mydiv <- div(class = "parent", id = "mother", "Not the mama!!!", "Hey!")
# fails
`[[`(mydiv$children, c(1, 2))
# works
`[`(mydiv$children, c(1, 2))
```

Alternatively, we could also create a `tagRemoveChildren` function. Also notice that the function raises an error if the provided tag does not have children.

5.5.3 Other interesting functions

The Golem package written by thinkr contains neat functions to edit your tags.

Particularly, the `tagRemoveAttributes`:

```
tagRemoveAttributes <- function(tag, ...) {
  attrs <- as.character(list(...))
  for (i in seq_along(attrs)) {
    tag$attribs[[ attrs[i] ]] <- NULL
  }
  tag
}
```

```
mydiv <- div(class = "test", id = "coucou", "Hello")
tagRemoveAttributes(mydiv, "class", "id")
```

5.5.4 Conditionally set attributes

Sometimes, you only want to set attributes under specific conditions.

```
my_button <- function(color = NULL) {
  tags$button(
    style = paste("color:", color),
    p("Hello")
  )
}

my_button()
```

This example will not fail but having `style="color: "` is not clean. We may use conditions:

```
my_button <- function(color = NULL) {
  tags$button(
    style = if (!is.null(color)) paste("color:", color),
    p("Hello")
  )
}

my_button("blue")
my_button()
```

In this example, style won't be available if color is not specified.

5.5.5 Using %>%

While doing a lot of manipulation for a tag, if you don't need to create intermediate objects, this is a good idea to use `%>%` from magrittr:

```
div(class = "cl", h1("Hello")) %>%
  tagAppendAttributes(id = "myid") %>%
  tagAppendChild(p("some extra text here!"))
```

The pipe syntax is overall easier to follow and read.

5.5.6 Programmatically create children elements

Assume you want to create a tag with three children inside:

```
div(
  span(1),
  span(2),
  span(3),
  span(4),
  span(5)
)
```

The structure is correct but imagine if you had to create 1000 `span` or fancier tag. The previous approach is not consistent with DRY programming. `lapply` function will be useful here (or the purrr `map` family):

```
# base R
div(lapply(1:5, function(i) span(i)))
# purrr + %>%
map(1:5, function(i) span(i)) %>% div()
```

Chapter 6

Dependency utilities

When creating a new template, you sometimes need to import custom HTML dependencies are not available in shiny. Fortunately, this is not aproblem using htmltools!

6.1 The dirty approach

This approach is dirty since it is not easily re-usable by others. Instead, we prefer a packaging approach, like in the next section.

Let's consider the following example. I want to include a bootstrap 4 card in a shiny app. Briefly, Bootstrap is the most popular HTML/CSS/JS framework to develop websites and web apps. This example is taken from an interesting question here. The naive approach would be to include the HTML code directly in the app code

```
# we create the card function before
my_card <- function(...) {
  withTags(
    div(
      class = "card border-success mb-3",
      div(class = "card-header bg-transparent border-success"),
      div(
        class = "card-body text-success",
        h3(class = "card-title", "title"),
        p(class = "card-text", ...)
      ),
      div(class = "card-footer bg-transparent border-success", "footer")
    )
```

```
  )
}

# we build our app
shinyApp(
  ui = fluidPage(
    fluidRow(
      column(
        width = 6,
        align = "center",
        br(),
        my_card("blablabla. PouetPouet Pouet.")
      )
    )
  ),
  server = function(input, output) {}
)
```

and unfortunately see that nothing is displayed. If you remember, this was expected since shiny does not contain bootstrap 4 dependencies and this card is unfortunately a bootstrap 4 object. Don't panic! Load the necessary css to display this card (if required, we could include the javascript as well). We could use either `includeCSS()`, `tags$head(tags$link(rel = "stylesheet", type = "text/css", href = "custom.css"))`. See more here.

```
shinyApp(
  ui = fluidPage(
    # load the css code
    includeCSS(path = "https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.m
    fluidRow(
      column(
        width = 6,
        align = "center",
        br(),
        my_card("blablabla. PouetPouet Pouet.")
      )
    )
  ),
  server = function(input, output) {}
)
```

The card may seem ugly but at least it is displayed. Fear not, we will fix the aesthetics later.

6.2 The clean approach

We will use the `htmlDependency` and `attachDependencies` functions from htmltools. The htmlDependency takes several arguments:

- the name of your dependency
- the version (useful to remember on which version it is built upon)
- a path to the dependency (can be a CDN or a local folder)
- script and stylesheet to respectively pass css and scripts

```
# handle dependency
card_css <- "bootstrap.min.css"
bs4_card_dep <- function() {
  htmlDependency(
    name = "bs4_card",
    version = "1.0",
    src = c(href = "https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/"),
    stylesheet = card_css
  )
}
```

We create the card tag and give it the bootstrap 4 dependency through the `attachDependencies()` function. In recent version of htmltools, we may simply use `tagList(tag, deps)` instead.

```
# create the card
my_card <- function(...) {
  cardTag <- withTags(
    div(
      class = "card border-success mb-3",
      div(class = "card-header bg-transparent border-success"),
      div(
        class = "card-body text-success",
        h3(class = "card-title", "title"),
        p(class = "card-text", ...)
      ),
      div(class = "card-footer bg-transparent border-success", "footer")
    )
  )

  # attach dependencies (old way)
  # htmltools::attachDependencies(cardTag, bs4_card_dep())

  # simpler way
  tagList(cardTag, bs4_card_dep())
```

```
}
```

We finally run our app:

```
# run shiny app
ui <- fluidPage(
  title = "Hello Shiny!",
  fluidRow(
    column(
      width = 6,
      align = "center",
      br(),
      my_card("blablabla. PouetPouet Pouet.")
    )
  )
)

shinyApp(ui, server = function(input, output) { })
```

With this approach, you could develop a package of custom dependencies that people could use when they need to add custom elements in shiny.

6.3 Another example: Importing HTML dependencies from other packages

The shinydashboard package helps to design dashboards with shiny. In the following, we would like to integrate the box component in a classic Shiny App (without the dashboard layout). However, if you try to include the Shinydashboard box tag, you will notice that nothing is displayed since Shiny does not have shinydashboard dependencies. Fortunately htmltools contains a function, namely `findDependencies` that looks for all dependencies attached to a tag. Before going futher, let's define the basic skeleton of a shinydashboard:

```
shinyApp(
  ui = dashboardPage(
    dashboardHeader(),
    dashboardSidebar(),
    dashboardBody(),
    title = "Dashboard example"
  ),
  server = function(input, output) { }
)
```

There are numerous details associated with shinydashboard that we will unfortunately not go into. If you are interested in learning more, please help yourself. The key point here is the main wrapper function dashboardPage. The fluidPage is another wrapper function that most are already familiar with. We apply `findDependencies` on `dashboardPage`.

```
deps <- findDependencies(
  dashboardPage(
    header = dashboardHeader(),
    sidebar = dashboardSidebar(),
    body = dashboardBody()
  )
)
deps
```

deps is a list containing four dependencies:

- Font Awesome handles icons
- Bootstrap is the main HTML/CSS/JS template. Importantly, please note the version 3.3.7, whereas the current is 4.3.1
- AdminLTE is the dependency containg HTML/CSS/JS related to the admin template. It is closely linked to Bootstrap 3.
- shinydashboard, the CSS and javascript necessary for shinydashboard to work properly. In practice, integrating custom HTML templates to shiny does not usually work out of the box for many reasons (Explain why!) and some modifications are necessary.

```
[[1]]
List of 10
$ name      : chr "font-awesome"
$ version   : chr "5.3.1"
$ src       :List of 1
..$ file: chr "www/shared/fontawesome"
$ meta      : NULL
$ script    : NULL
$ stylesheet: chr [1:2] "css/all.min.css" "css/v4-shims.min.css"
$ head      : NULL
$ attachment: NULL
$ package   : chr "shiny"
$ all_files : logi TRUE
- attr(*, "class")= chr "html_dependency"
[[2]]
List of 10
$ name      : chr "bootstrap"
$ version   : chr "3.3.7"
```

```
$ src       :List of 2
..$ href: chr "shared/bootstrap"
..$ file: chr "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/shiny/ww
$ meta      :List of 1
..$ viewport: chr "width=device-width, initial-scale=1"
$ script    : chr [1:3] "js/bootstrap.min.js" "shim/html5shiv.min.js" "shim/respond.mi
$ stylesheet: chr "css/bootstrap.min.css"
$ head      : NULL
$ attachment: NULL
$ package   : NULL
$ all_files : logi TRUE
- attr(*, "class")= chr "html_dependency"
[[3]]
List of 10
$ name      : chr "AdminLTE"
$ version   : chr "2.0.6"
$ src       :List of 1
..$ file: chr "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/shinydas
$ meta      : NULL
$ script    : chr "app.min.js"
$ stylesheet: chr [1:2] "AdminLTE.min.css" "_all-skins.min.css"
$ head      : NULL
$ attachment: NULL
$ package   : NULL
$ all_files : logi TRUE
- attr(*, "class")= chr "html_dependency"
[[4]]
List of 10
$ name      : chr "shinydashboard"
$ version   : chr "0.7.1"
$ src       :List of 1
..$ file: chr "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/shinydas
$ meta      : NULL
$ script    : chr "shinydashboard.min.js"
$ stylesheet: chr "shinydashboard.css"
$ head      : NULL
$ attachment: NULL
$ package   : NULL
$ all_files : logi TRUE
- attr(*, "class")= chr "html_dependency"
```

Below, we attach the dependencies to the `box` with `tagList`, as shown above. Notice that our custom `box` does not contain all parameters from shinydashboard, which is actually ok at this time.

```
my_box <- function(title, status) {
  tagList(box(title = title, status = status), deps)
}
ui <- fluidPage(
  titlePanel("Shiny with a box"),
  my_box(title = "My box", status = "danger"),
)
server <- function(input, output) {}
shinyApp(ui, server)
```

You now have limitless possibilities! Interestingly, this same approach is the basis of shinyWidgets for the `useBs4Dash` function and other related tools.

6.4 Suppress dependencies

In rare cases, you may need to remove an existing conflicting dependency. The `suppressDependencies` function allows users to perform this. For instance, shiny.semantic built on top of semantic ui is not compatible with Bootstrap. Below, we remove the AdminLTE dependency from a shinydashboard page and nothing is displayed (as expected):

```
shinyApp(
  ui = dashboardPage(
    dashboardHeader(),
    dashboardSidebar(),
    dashboardBody(suppressDependencies("AdminLTE")),
    title = "Dashboard example"
  ),
  server = function(input, output) { }
)
```

Chapter 7

Other tools

7.1 CSS

- See cascadess to customize the style of tags

```
ui <- list(
  cascadess(),
  h4(
    .style %>%
      font(case = "upper") %>%
      border(bottom = "red"),
    "Etiam vel tortor sodales tellus ultricies commodo."
  )
)
```

Practice

In this chapter, you will learn how to build your own html templates taken from the web and package them, so that they can be re-used at any time by anybody.

Chapter 8

Template selection

There exists tons of HTML templates over the web. However, only a few part will be suitable for shiny, mainly because of what follows:

- shiny is built on top of Bootstrap 3 (HTML, CSS and Javascript framework), meaning that changing the framework will not be trivial. However, shinymaterial and shiny.semantic are good examples that show this is possible.
- shiny relies on jQuery (currently v 3.4.1 for shiny). Consequently, all templates based upon React, Vue and other Javascript framework will not be natively supported. Again, there exist some examples for React with shiny and more generally, the reactR package developed by Kent Russell and Alan Dipert from RStudio.

See the github repository for more details about all dependencies related to the shiny package.

> Notes: As shiny depends on Bootstrap 3.4.1, we recommend the user whom is interested in experimenting with Bootstrap 4 to be consciously aware of the potential incompatibilities to be particularly careful about potential incompatibilies. See a working example here with bs4Dash.

A good source of **open source** HTML templates is Colorlib and Creative Tim.

In the next chapter, we will focus on the tabler.io dashboard template (See Figure 8.1).

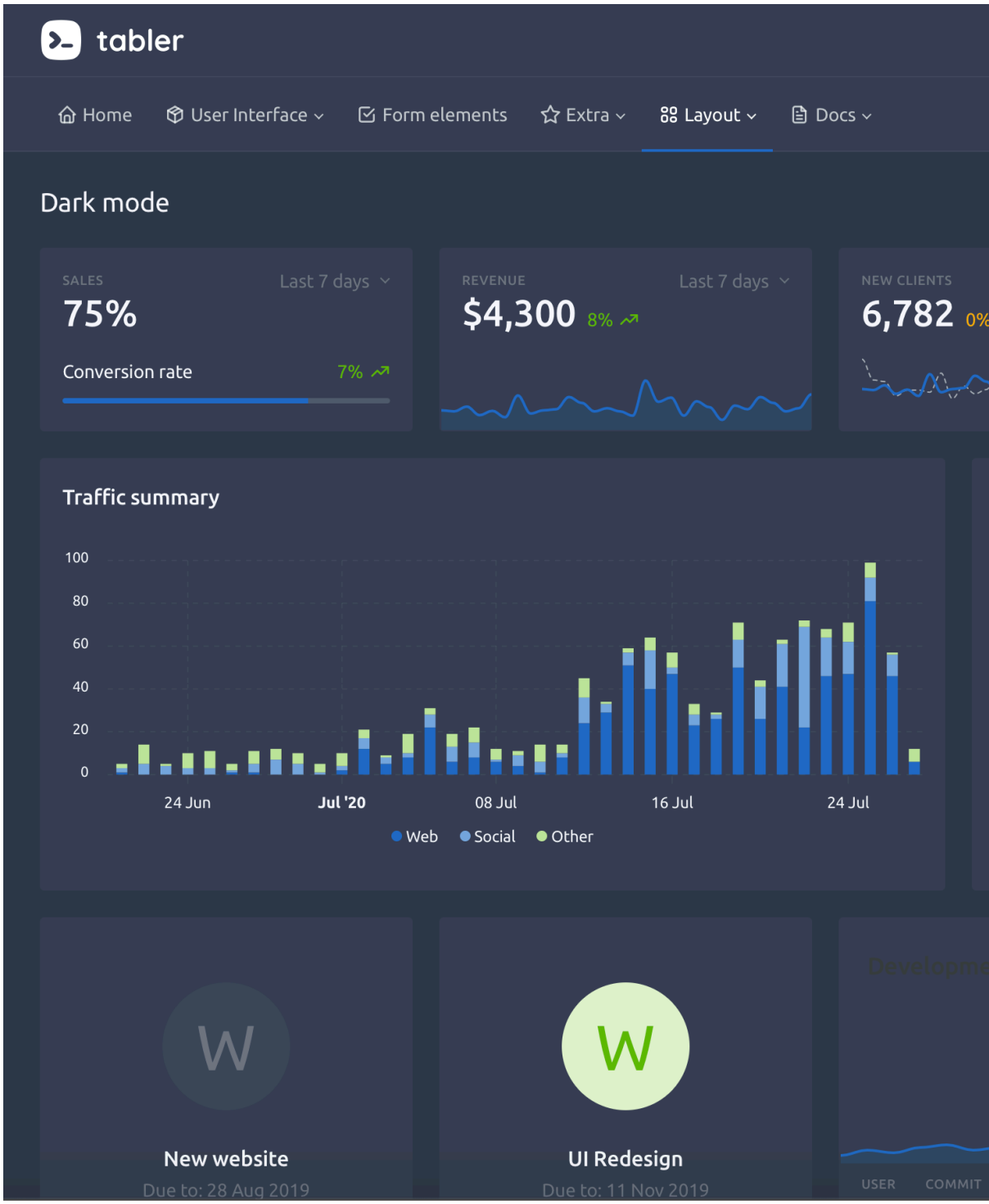


Figure 8.1: Tabler dashboard overview

Chapter 9

Define dependencies

This is time to start our practical session. As mentionned in the previous chapter, we will focus on the tabler, a tiny Bootstrap 4 dashboard template. Importantly, what follows is not the description of how to customize tabler but rather provide a R wrapper. Therefore we will not write SASS nore edit the core JavaScript, even though technically possible.

9.1 Discover the project

The first step of any template adaptation consists in exploring the underlying github repository (if open source) and look for mandatory elements, like CSS/JS dependencies. You would actually proceed similarly for an HTMLWidget.

As shown in Figure 9.1, the most important folders are:

- dist: contains CSS and JS files as well as other libraries like Bootstrap and jQuery. In production we seek for minified files since they take less space. It is also a good moment to look at the version of each dependency that might conflict with Shiny
- demo is the website folder used for demonstration purpose. This is our source to explore the template capabilities in depth

The scss and build folder are also crucial to the package but as stated previously, customizing tabler is out of the scope of this book. It is already a significant task to adapt a template from a language to another.

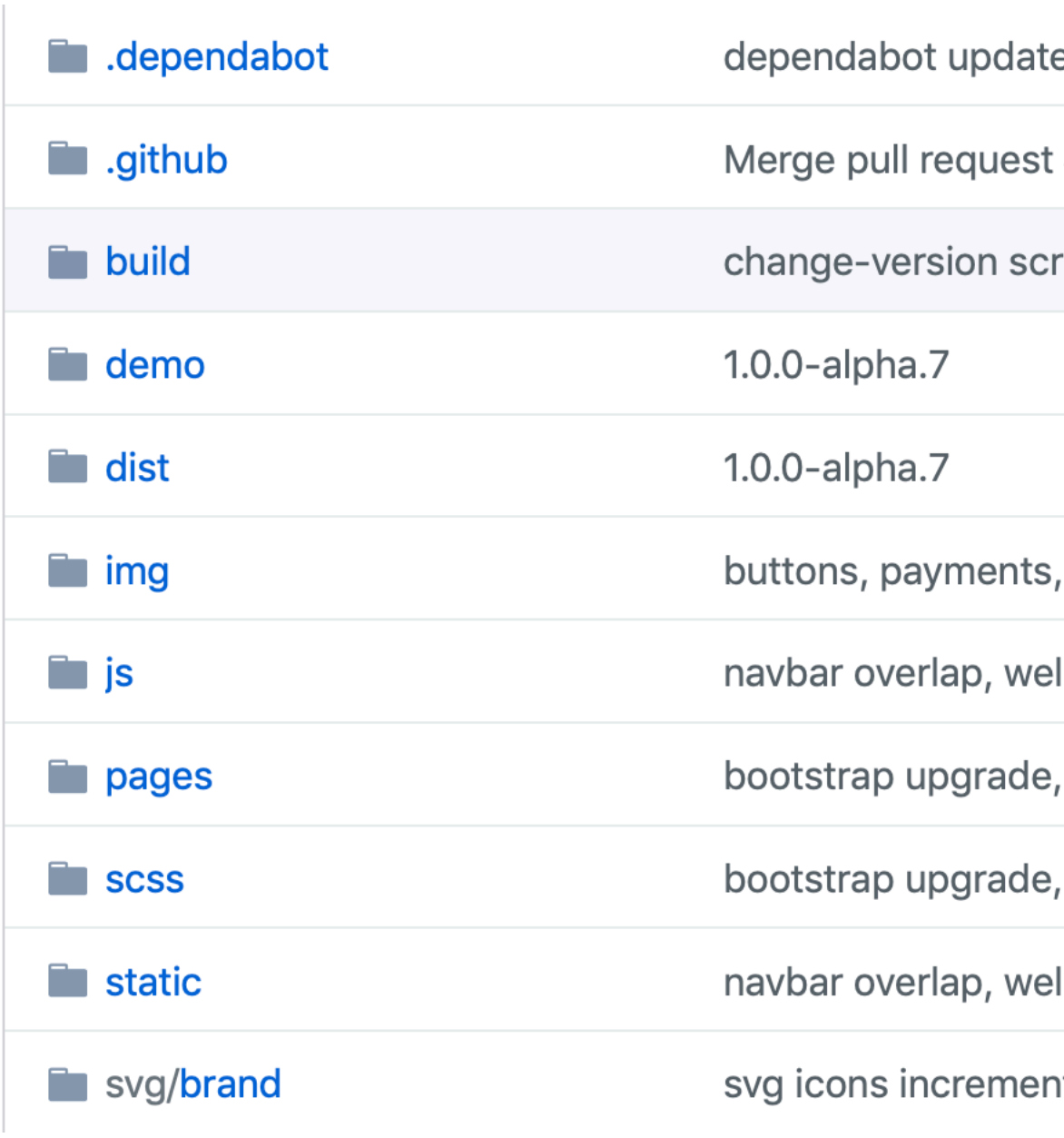


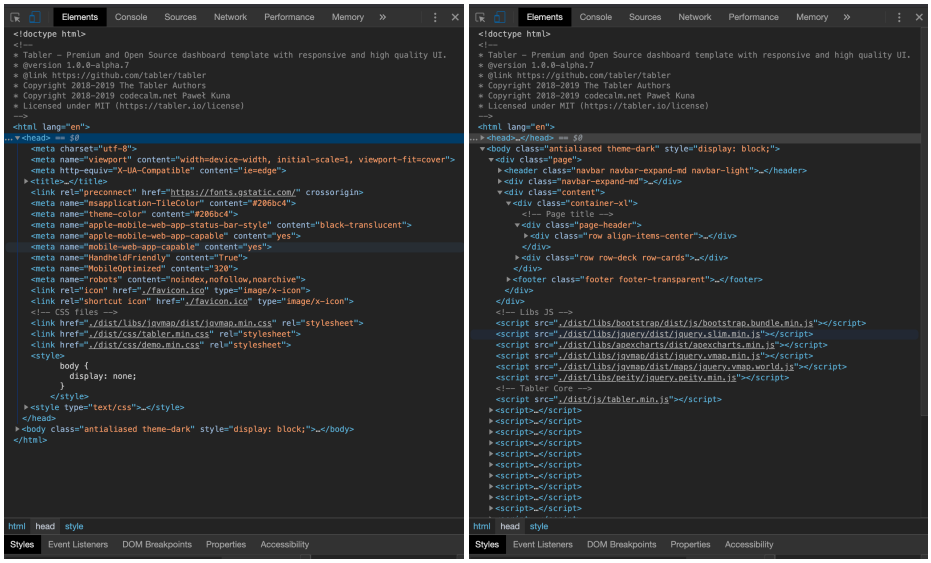
Figure 9.1: Github project exploration

9.2 Identify mandatory dependencies

Now, among all JS/CSS resources, we need to identify the one necessary to the template. Obviously, the Bootstrap 4, jQuery, tabler.min.css and tabler.min.js are key elements, contrary to flag icons which are optional (and take a lot of space). The package size is also to consider if you plan to release the template on CRAN and respect the 5mB maximum limit. By experience, I can tell you this is quite hard to handle.

To inspect dependencies, we proceed as follows

- Download or clone the github repository
- Go to the demo folder and open the layout-dark.html file
- Open the HTML inspector



As depicted on Figure **??** left-hand side, we need to include the tabler.min.css from the header. If you are not convinced, try to remove it from the DOM and see what happens. jqvmap is actually related to an external visualization plugin used in the demo. Finally the demo.min.css file is for the demo purpose. This will not prevent the template to work if we skip it. So far so good, we only need 1 file thus!

JavaScript dependencies are shown on the right-hand side and located at the end of the body tag. We won't need all chart-related dependencies like apexcharts, jquery.vmap and vmap world and can sefely ignore them. We will keep the Bootstrap 4 bundle.js, jQuery core and tabler.min.js (the order is crucial).

9.3 Bundle dependencies

With the help of the `htmltoolsDependency` function, we are going to create our main tabler HTML dependency containing all assets to allow our template to render properly. In this example, I am going to cheat a bit: instead of handling local files, I will use a CDN (content delivery network) that hosts all necessary tabler assets. The main reason is that it will allow us to test the template directly from the bookdown project. But in theory, this template would need to live inside an R package, in a github repository.

```
tablers_deps <- htmlDependency(
  name = "tabler",
  version = "1.0.7", # we take that of tabler,
  src = c(href = "https://cdn.jsdelivr.net/npm/tabler@1.0.0-alpha.7/dist/"),
  script = "js/tabler.min.js",
  stylesheet = "css/tabler.min.css"
)
```

I advise the reader to create 1 HTML dependency per element. The Bootstrap version is v4.3.1 (Shiny relies on 3.4.1) and jQuery is 3.5.0 (Shiny relies on 3.4.1). We can also use a CDN:

```
bs4_deps <- htmlDependency(
  name = "Bootstrap",
  version = "4.3.1",
  src = c(href = "https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/"),
  script = "bootstrap.bundle.min.js"
)

jQuery_deps <- htmlDependency(
  name = "jquery",
  version = "3.5.0",
  src = c(href = "https://code.jquery.com/"),
  script = "jquery-3.5.0.slim.min.js"
)
```

We finally create our dependency manager (TO DO: add more details):

```
# add all dependencies to a tag. Don't forget to set append to TRUE to preserve any ex
addDeps <- function(tag) {
  # below, the order is of critical importance!
  deps <- list(bs4_deps, tablers_deps)
  attachDependencies(tag, deps, append = TRUE)
}
```

Let's see how to use `addDeps`. We consider a `<div>` placeholder and check for its dependencies with `findDependencies` (should be NULL). Then, we wrap it with `addDeps`.

```
tag <- div()
findDependencies(tag)
```

```
## NULL
```

```
tag <- addDeps(div())
findDependencies(tag)
```

```
## [[1]]
## List of 10
##  $ name      : chr "Bootstrap"
##  $ version   : chr "4.3.1"
##  $ src       :List of 1
##   ..$ href: chr "https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/"
##  $ meta      : NULL
##  $ script    : chr "bootstrap.bundle.min.js"
##  $ stylesheet: NULL
##  $ head      : NULL
##  $ attachment: NULL
##  $ package   : NULL
##  $ all_files : logi TRUE
##  - attr(*, "class")= chr "html_dependency"
##
## [[2]]
## List of 10
##  $ name      : chr "tabler"
##  $ version   : chr "1.0.7"
##  $ src       :List of 1
##   ..$ href: chr "https://cdn.jsdelivr.net/npm/tabler@1.0.0-alpha.7/dist/"
##  $ meta      : NULL
##  $ script    : chr "js/tabler.min.js"
##  $ stylesheet: chr "css/tabler.min.css"
##  $ head      : NULL
##  $ attachment: NULL
##  $ package   : NULL
##  $ all_files : logi TRUE
##  - attr(*, "class")= chr "html_dependency"
```

As shown above, our dependencies are applied to the div, in the correct order. This order is set by the list `list(bs4_deps, jQuery_deps, tablers_deps)`.

This flexibility allows to avoid potential conflicts. If we try to run this simple tag in a shiny app, we notice that all dependencies are added to the `<head>` tag, whereas the original template loads JavaScript dependencies in the `<body>`. Currently, htmltools does not allow to distribute dependencies in different places. Here there is no impact but for other templates like Framework7 (which is powering shinyMobile), JavaScript must be place in the body. In practice, this is quite honestly hard to guess and only manual testing will help you.

```
ui <- fluidPage(tag)
server <- function(input, output, session) {}
shinyApp(ui, server)
```

Even though the `addDeps` function may be applied to any tag, we will use it with the core HTML template, that remain to be designed!

Would you like to see if our dependency system works? Let's meet in the next chapter to design the main dashboard layout.

Chapter 10

Template skeleton

The list of all available layouts is quite impressive (horizontal, vertical, compressed, right to left, dark, …). In the next steps, we will focus on the dark-compressed template. We leave the reader to try other templates as an exercise.

10.1 Identify template elements

We are quite lucky since there is nothing fancy about the tabler layout. As usual, let's inspect the layout-condensed-dark.html (in the tabler /demo folder) in Figure 10.1

There are 2 main components: - the header containing the brand logo, the navigation and dropdown - the content containing the dashboard body as well as the footer

Something important: the dashboard body does not mean `<body>` tag!

This is all!

10.2 Design the page layout

10.2.1 The page wrapper

Do you remember the structure of a basic html page seen in Chapter 2? Well, if not, here is a reminder.

```
<!DOCTYPE HTML>
<html>
```

```
<!doctype html>
<!--
* Tabler - Premium and Open Source dashboard template with responsive
* @version 1.0.0-alpha.7
* @link https://github.com/tabler/tabler
* Copyright 2018-2019 The Tabler Authors
* Copyright 2018-2019 codecalm.net Paweł Kuna
* Licensed under MIT (https://tabler.io/license)
-->
<html lang="en">
... ▶ <head>…</head> == $0
  ▼ <body class="antialiased" style="display: block;">
    ▼ <div class="page">
      ▼ <header class="navbar navbar-expand-md navbar-dark">
        ▼ <div class="container-xl">
          ▶ <button class="navbar-toggler" type="button" data-toggle="col
          "#navbar-menu">…</button>
          ▶ <a href="." class="navbar-brand navbar-brand-autodark d-none-
          pr-md-3">…</a>
          ▶ <div class="navbar-nav flex-row order-md-last">…</div>
          ▼ <div class="collapse navbar-collapse" id="navbar-menu">
            ▼ <div class="d-flex flex-column flex-md-row flex-fill align-
            items-md-center">
              ▶ <ul class="navbar-nav">…</ul>
              ▶ <div class="ml-md-auto pl-md-4 py-2 py-md-0 mr-md-4 order-
              flex-grow-1 flex-md-grow-0">…</div>
              </div>
            </div>
          </div>
        </header>
      ▼ <div class="content">
        ▼ <div class="container-xl">
            <!-- Page title -->
          ▶ <div class="page-header">…</div>
          ▶ <div class="row row-deck row-cards">…</div>
          </div>
        ▶ <footer class="footer footer-transparent">…</footer>
        </div>
      </div>
```

Figure 10.1: Tabler condensed layout

```
  <head>
  <!-- head content here -->
  </head>
  <body>
    <p>Hello World</p>
  </body>
</html>
```

We actually don't need to take care of the `<html>` tag. Below we construct a list of tags with `tagList`, including the head and the body. In the head we have `meta` tag that briefly describe the encoding, how to display the app on different devices (For instance `apple-mobile-web-app-status-bar-style` is for mobile support), set the favicon (website icon, the icon you see on the right side of the searchbar. Try twitter for instance). The page title may change, so is the favicon, so we include them as parameters of the function. If you remember, there also should be CSS in the head but nothing here! Actually, the insertion of dependencies will be achieved by our `addDeps` function defined in Chapter 9. This is what we do to the `<body>` tag that is wrapped by this function. Let's talk about the dark parameter. In short, the only difference between the dark and the light theme is the class applied to the `<body>` tag (respectively "antialiased theme-dark" and "antialiased"). The … parameter contain other template elements like the header and the dashboard body, that remain to be created.

```
tabler_page <- function(..., dark = TRUE, title = NULL, favicon = NULL){

  tagList(
    # Head
    tags$head(
      tags$meta(charset = "utf-8"),
      tags$meta(
        name = "viewport",
        content = "
        width=device-width,
        initial-scale=1,
        viewport-fit=cover"
      ),
      tags$meta(`http-equiv` = "X-UA-Compatible", content = "ie=edge"),
      tags$title(title),
      tags$link(
        rel = "preconnect",
        href = "https://fonts.gstatic.com/",
        crossorigin = NA
      ),
      tags$meta(name = "msapplication-TileColor", content = "#206bc4"),
```

```
      tags$meta(name = "theme-color", content = "#206bc4"),
      tags$meta(name = "apple-mobile-web-app-status-bar-style", content = "black-transl
      tags$meta(name = "apple-mobile-web-app-capable", content = "yes"),
      tags$meta(name = "mobile-web-app-capable", content = "yes"),
      tags$meta(name = "HandheldFriendly", content = "True"),
      tags$meta(name = "MobileOptimized", content = "320"),
      tags$meta(name = "robots", content = "noindex,nofollow,noarchive"),
      tags$link(rel = "icon", href = favicon, type = "image/x-icon"),
      tags$link(rel = "shortcut icon", href = favicon, type="image/x-icon")
    ),
    # Body
    addDeps(
      tags$body(
        tags$div(
          class = paste0("antialiased ", if(dark) "theme-dark"),
          style = "display: block;",
          tags$div(class = "page", ...)
        )
      )
    )
  )
}
```

Below we quickly test if a tabler element renders well to see whether our dependency system is adequately setup. To that end, we include a random tabler element taken from the demo html page and include it as raw html, using `HTML`. We also ensure that basic Shiny input/output system works as expected with a sliderInput linked to a plot output.

```
thematic_on()
onStop(thematic_off)
ui <- tabler_page(
  "test",
  sliderInput("obs", "Number of observations:",
              min = 0, max = 1000, value = 500
  ),
  plotOutput("distPlot"),
  br(),
  HTML(
    '<div class="col-sm-6 col-lg-3">
   <div class="card">
      <div class="card-body">
         <div class="d-flex align-items-center">
            <div class="subheader">Sales</div>
            <div class="ml-auto lh-1">
```

```
                <div class="dropdown">
                   <a class="dropdown-toggle text-muted" href="#" data-toggle="dropdown" aria-hasp
                   Last 7 days
                   </a>
                   <div class="dropdown-menu dropdown-menu-right">
                      <a class="dropdown-item active" href="#">Last 7 days</a>
                      <a class="dropdown-item" href="#">Last 30 days</a>
                      <a class="dropdown-item" href="#">Last 3 months</a>
                   </div>
                </div>
             </div>
          </div>
          <div class="h1 mb-3">75%</div>
          <div class="d-flex mb-2">
             <div>Conversion rate</div>
             <div class="ml-auto">
                <span class="text-green d-inline-flex align-items-center lh-1">
                   7%
                   <svg xmlns="http://www.w3.org/2000/svg" class="icon ml-1" width="24" height="24
                      <path stroke="none" d="M0 0h24v24H0z"></path>
                      <polyline points="3 17 9 11 13 15 21 7"></polyline>
                      <polyline points="14 7 21 7 21 14"></polyline>
                   </svg>
                </span>
             </div>
          </div>
          <div class="progress progress-sm">
             <div class="progress-bar bg-blue" style="width: 75%" role="progressbar" aria-valuenow
                <span class="sr-only">75% Complete</span>
             </div>
          </div>
       </div>
    </div>
</div>
    '
  ),
title = "Tabler test"
)
server <- function(input, output, session) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs))
  })
}
shinyApp(ui, server)
```

Ok… The layout is ugly, margins are not correct, the plot background does not match with the overall theme, … but our info card and the shiny element work like a charm, which is a good start.

10.2.2 The body content

In this part, we translate the dashboard body HTML code to R. We create a function called `tabler_body`. The … parameter holds all the dashboard body elements and the footer is dedicated for the future `tabler_footer` function.

```
tabler_body <- function(..., footer = NULL) {
  div(
    class = "content",
    div(class = "container-xl", ...),
    tags$footer(class = "footer footer-transparent", footer)
  )
}
```

Let's test it with the previous example.

```
ui <- tabler_page(tabler_body(p("Hello World")))
server <- function(input, output, session) {}
shinyApp(ui, server)
```

Way better!

10.2.3 The footer

The footer is composed of a left and right containers. We decide to create parameters left and right in which the user will be able to pass any elements.

```
tabler_footer <- function(left = NULL, right = NULL) {
  div(
    class = "container",
    div(
      class = "row text-center align-items-center flex-row-reverse",
      div(class = "col-lg-auto ml-lg-auto", right),
      div(class = "col-12 col-lg-auto mt-3 mt-lg-0", left)
    )
  )
}
```

As above, let's check our brand new element.

```
ui <- tabler_page(
  tabler_body(
    p("Hello World"),
    footer = tabler_footer(
      left = "Rstats, 2020",
      right = a(href = "https://www.google.com")
    )
  )
)
server <- function(input, output, session) {}
shinyApp(ui, server)
```

10.2.4 The navbar (or header)

This function is called `tabler_header`. In the tabler template, the header has the classes "navbar navbar-expand-md navbar-light". We don't need the navbar-light classe since we are interested in the dark theme. As shown in Figure 10.2, the navbar is composed of 4 elements:

- the navbar toggler is only visible when we reduce the screen width (like on mobile devices)
- the brand image
- the navigation
- the dropdown menu (this is not mandatory)

You may have a look at the Bootstrap 4 documentation for the navbar configuration and layout.

Each of these element will be considered as an input parameter to the `tabler_navbar` function, except the first element which is a default element and should not be removed. Morever, we will only show the brand element when it is provided. The … parameter is a slot for extra elements (between the menu and dropdowns).

```
tabler_navbar <- function(..., brand_url = NULL, brand_image = NULL, nav_menu, nav_right = NULL)
  navbar_cl <- "navbar navbar-expand-md"
  tags$header(
    class = navbar_cl,
    tags$div(
      class = "container-xl",
      # toggler for small devices (must not be removed)
      tags$button(
        class = "navbar-toggler",
        type = "button",
```

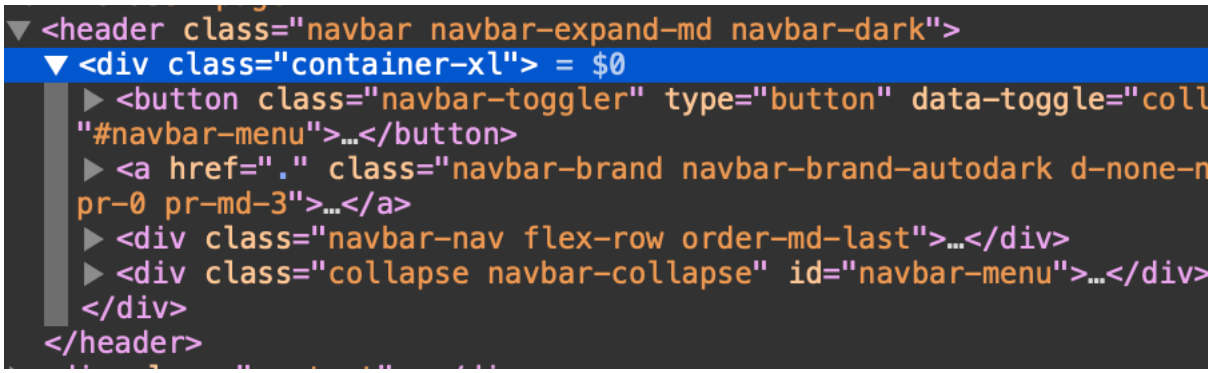


Figure 10.2: Tabler header structure

```
      `data-toggle` = "collapse",
      `data-target` = "#navbar-menu",
      span(class = "navbar-toggler-icon")
    ),

    # brand stuff
    if (!is.null(brand_url) || !is.null(brand_image)) {
      a(
        href = if (!is.null(brand_url)) {
          brand_url
        } else {
          "#"
        },
        class = "navbar-brand navbar-brand-autodark d-none-navbar-horizontal pr-0 pr
        if(!is.null(brand_image)) {
          img(
            src = brand_image,
            alt = "brand Image",
            class = "navbar-brand-image"
          )
        }
      )
    },

    # slot for dropdown element
    if (!is.null(nav_right)) {
      div(class = "navbar-nav flex-row order-md-last", nav_right)
    },

    #
```

```
      div(
        class = "collapse navbar-collapse",
        id = "navbar-menu",
        div(
          class = "d-flex flex-column flex-md-row flex-fill align-items-stretch align-items-md-ce
          nav_menu
        ),
        if (length(list(...)) > 0) {
          div(
            class = "ml-md-auto pl-md-4 py-2 py-md-0 mr-md-4 order-first order-md-last flex-grow-
            ...
          )
        }
      )
    )
  )
}
```

Let's create the navbar menu. The … parameter is a slot for the menu items. Compared to the original tabler dashboard template where there is only the class navbar-nav, we have to add, at least, the nav class to make sure items are correctly activated/inactivated. The nav-pills class is to select pills instead of basic tabs (see here).

```
tabler_navbar_menu <- function(...) {
  tags$ul(class = "nav nav-pills navbar-nav", ...)
}
```

Each navbar menu item could be either a simple button or contain multiple menu sub-items. For now, we only focus on simple items.

10.2.4.1 Navbar navigation

This part is extremely important since it will drives the navigation of the template. What do we want? We would like to associate each item to a separate page in the body content, so that each time we change item, we go on another page. In brief, it is very similar to the Shiny `tabsetPanel` function.

In HTML, menu items are `<a>` tags (links) with a given `href` attribute pointing to a specific page located in the server files. The point with a Shiny app is that we can't decide how to split our content into several pages. We only have app.R generating a simple HTML page. The strategy here is to create a tabbed navigation, to mimic multiple pages.

Let's see how tabset navigation works. In the menu list, all items must have a `data-toggle` attribute set to tab, an `href` attribute holding a unique id. This

unique id is mandatory since it will point the menu item to the corresponding body content. On the body side, tab panels are contained in a tabset panel (simple div container), have a `role` attribute set to tabpanel and an `id` corresponding the tabName passed in the menu item. Below, we propose a possible implementation of a menu item, as well as the corresponding body tab panel.

```
tabler_navbar_menu_item <- function(text, tabName, icon = NULL, selected = FALSE) {

  item_cl <- paste0("nav-link", if(selected) " active")

  tags$li(
    class = "nav-item",
    a(
      class = item_cl,
      href = paste0("#", tabName),
      `data-toggle` = "pill", # see https://getbootstrap.com/docs/4.0/components/navs/
      `data-value` = tabName,
      role = "tab",
      span(class = "nav-link-icon d-md-none d-lg-inline-block", icon),
      span(class = "nav-link-title", text)
    )
  )
}
```

We also decided to add a fade transition effect between tabs, as per Bootstrap 4 documentation.

```
tabler_tab_items <- function(...) {
  div(class = "tab-content", ...)
}

tabler_tab_item <- function(tabName = NULL, ...) {
  div(
    role = "tabpanel",
    class = "tab-pane fade container-fluid",
    id = tabName,
    ...
  )
}
```

What about testing this in a shiny app?

```
ui <- tabler_page(
  tabler_navbar(
    brand_url = "https://preview-dev.tabler.io",
```

```
    brand_image = "https://preview-dev.tabler.io/static/logo.svg",
    nav_menu = tabler_navbar_menu(
      tabler_navbar_menu_item(
        text = "Tab 1",
        icon = NULL,
        tabName = "tab1",
        selected = TRUE
      ),
      tabler_navbar_menu_item(
        text = "Tab 2",
        icon = NULL,
        tabName = "tab2"
      )
    )
  ),
  tabler_body(
    tabler_tab_items(
      tabler_tab_item(
        tabName = "tab1",
        p("Hello World")
      ),
      tabler_tab_item(
        tabName = "tab2",
        p("Second Tab")
      )
    ),
    footer = tabler_footer(
      left = "Rstats, 2020",
      right = a(href = "https://www.google.com")
    )
  )
)
server <- function(input, output, session) {}
shinyApp(ui, server)
```

10.2.4.2 Fine tune tabs behavior

Quite good isn't it? However, you will notice that even if the first tab is selected by default, its content is not shown! To fix this, we will apply our jQuery skills. According to the Bootstrap documentation, we must trigger the show event on the active tab at start, as well as add the classes show and active to the associated tab panel in the dashboard body. We therefore target the nav item that has the active class and if no item is found, we select the first item by default and activate its tab. We will explore another alternative in Chapter 11.

```
$(function() {
  // this makes sure to trigger the show event on the active   tab at start
  let activeTab = $('#navbar-menu .nav-link.active');
  // if multiple items are found
  if (activeTab.length > 0) {
    let tabId = $(activeTab).attr('data-value');
    $(activeTab).tab('show');
    $(`#${tabId}`).addClass('show active');
  } else {
    $('#navbar-menu .nav-link')
      .first()
      .tab('show');
  }
});
```

The script is included in a tag but best practice it to put it in a separate js file (I do it this way because it is more convenient for the demonstration).

```
thematic_on()
onStop(thematic_off)
ui <- tabler_page(
  tags$head(
    tags$script(
      HTML(
        "$(function() {
          // this makes sure to trigger the show event on the active tab at start
          const activeTab = $('#navbar-menu .nav-link.active');
          if (activeTab.length > 0) {
            const tabId = $(activeTab).attr('data-value');
            $(activeTab).tab('show');
            $(`#${tabId}`).addClass('show active');
          } else {
            $('#navbar-menu .nav-link')
              .first()
              .tab('show');
          }
        });
        "
      )
    )
  ),
  tabler_navbar(
    brand_url = "https://preview-dev.tabler.io",
    brand_image = "https://preview-dev.tabler.io/static/logo.svg",
    nav_menu = tabler_navbar_menu(
```

```
      tabler_navbar_menu_item(
        text = "Tab 1",
        icon = NULL,
        tabName = "tab1",
        selected = TRUE
      ),
      tabler_navbar_menu_item(
        text = "Tab 2",
        icon = NULL,
        tabName = "tab2"
      )
    )
  ),
  tabler_body(
    tabler_tab_items(
      tabler_tab_item(
        tabName = "tab1",
        sliderInput(
          "obs",
          "Number of observations:",
          min = 0,
          max = 1000,
          value = 500
        ),
        plotOutput("distPlot")
      ),
      tabler_tab_item(
        tabName = "tab2",
        p("Second Tab")
      )
    ),
    footer = tabler_footer(
      left = "Rstats, 2020",
      right = a(href = "https://www.google.com")
    )
  )
)
server <- function(input, output, session) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs))
  })
}
shinyApp(ui, server)
```

The result is shown in Figure 10.3. I'd also suggest to include at least 1 in-

put/output per tab, to test whether everything works properly.

Looks like we are done for the main template elements. Actually, wouldn't it be better to include, at least, card containers?

10.2.5 Card containers

Card are a central piece of template as they may contain visualizations, metrics and much more. Tabler has a large range of card containers.

10.2.5.1 Classic card

What I call a classic card is like the `box` container of shinydashboard. The card structure has key elements:

- a width to control the space taken by the card in the Bootstrap grid
- a title, in general in the header (tabler does always not follow this rule and header is optional)
- a body where is the main content
- style elements like color statuses
- a footer (optional, tabler does not include this)

A comprehensive list of all tabler card features may be found here. To be faster, I will copy the following HTML code in the html2R shiny app to convert it to Shiny tags

```
<div class="col-md-6">
  <div class="card">
    <div class="card-status-top bg-danger"></div>
    <div class="card-body">
      <h3 class="card-title">Title</h3>
      <p>Some Text.</p>
    </div>
  </div>
</div>
```

Below is the result. The next step consist in replacing all content by parameters to the `tabler_card` function, whenever necessary. For instance, the first `<div>` sets the width of the card. The Bootstrap grid ranges from 0 to 12, so why not creating a width parameter to control the card size. We proceed similarly for the title, status, body content. A last comment on parameters default values. It seems reasonable to allow title to be NULL (if so, the title won't be shown), same thing for the status. Regarding the card default width, 6 also makes sense.

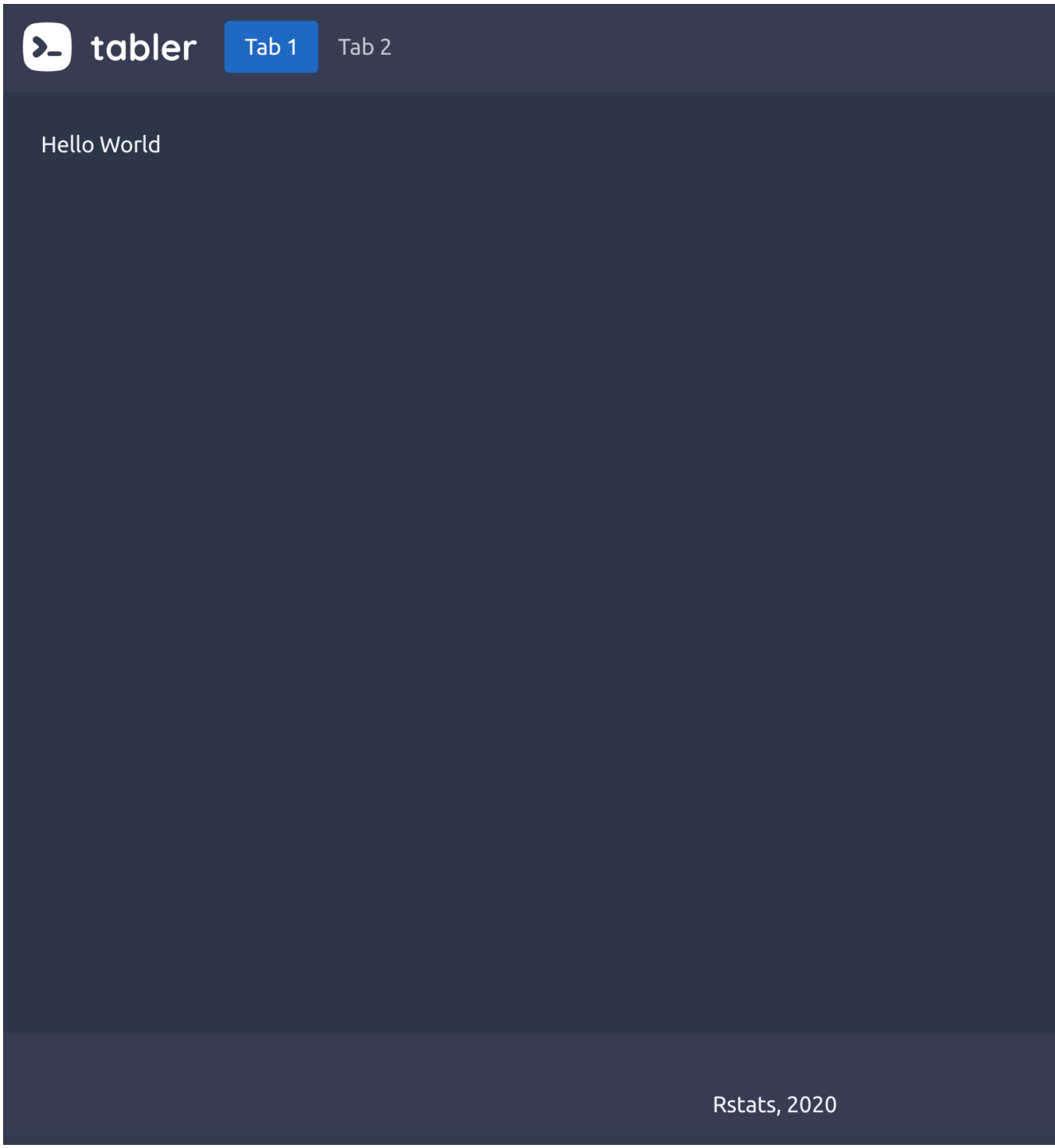


Figure 10.3: Tabler template with navbar

```
tabler_card <- function(..., title = NULL, status = NULL, width = 6, stacked = FALSE, p

  card_cl <- paste0(
    "card",
    if (stacked) " card-stacked",
    if (!is.null(padding)) paste0(" card-", padding)
  )

  div(
    class = paste0("col-md-", width),
    div(
      class = card_cl,
      if (!is.null(status)) {
        div(class = paste0("card-status-top bg-", status))
      },
      div(
        class = "card-body",
        # we could have a smaller title like h4 or h5...
        if (!is.null(title)) {
          h3(class = "card-title", title)
        },
        ...
      )
    )
  )

}

# test the card
my_card <- tabler_card(
  p("Hello"),
  title = "My card",
  status = "danger"
)
```

In the meantime, I'd be also nice to be able to display cards in the same row. Let's create the `tabler_row`:

```
tabler_row <- function(...) {
  div(class = "row row-deck", ...)
}
```

```
ui <- tabler_page(
  tabler_body(
    tabler_row(
```

```
      my_card,
      tabler_card(
        p("Hello"),
        title = "My card",
        status = "success"
      )
    )
  )
)
server <- function(input, output, session) {}
shinyApp(ui, server)
```

10.2.6 Ribbons: card components

Let's finish this part by including a card component, namely the ribbon.

```
tabler_ribbon <- function(..., position = NULL, color = NULL, bookmark = FALSE) {

  ribbon_cl <- paste0(
    "ribbon",
    if (!is.null(position)) sprintf(" bg-%s", position),
    if (!is.null(color)) sprintf(" bg-%s", color),
    if (bookmark) " ribbon-bookmark"
  )
  div(class = ribbon_cl, ...)
}
```

Integrating the freshly created ribbon component requires to modify the card structure since the ribbon is added after the body tag, and not parameter is associated with this slot. We could also modify the `tabler_card` function but htmltools contains tools to help us. Since the ribbon should be put after the card body (but in the card container), we may think about the `tagAppendChild` function, introduced in Chapter 5:

```
# add the ribbon to a card
my_card <- tabler_card(title = "Ribbon")
my_card$children[[1]] <- my_card$children[[1]] %>%
  tagAppendChild(
    tabler_ribbon(
      icon("info-circle", class = "fa-lg"),
      bookmark = TRUE
    )
  )
```

As shown above, the ribbon has been successfuly included in the card tag. Now, we check how it looks in a shiny app.

```
ui <- tabler_page(
  tabler_body(
    my_card
  )
)
server <- function(input, output, session) {}
shinyApp(ui, server)
```

10.2.7 Icons

Not mentionned before but we can use fontawesome icons provided with Shiny, as well as other libraries. Moreover, tabler has a svg library located here.

Chapter 11

Develop custom input widgets

In the previous chapter, we built the template dependencies, the page skeleton as well as containers like cards. However, it would be nice to custonize user interactions by bringing new inputs. In this chapter, we will apply knowledge from Chapter 4 about creating new Shiny input.

11.1 Tabler action button

Let's start with a simple input: the action button. Tabler has builtin HTML buttons with a substantial amount of custom styles, compared to the classic Shiny action button.

11.1.1 Reminders about the action button

Below is the code of the `actionButton` input.

```
actionButton <- function (inputId, label, icon = NULL, width = NULL, ...)  {
  value <- restoreInput(id = inputId, default = NULL)
  tags$button(
    id = inputId,
    style = if (!is.null(width)) paste0("width: ", validateCssUnit(width), ";"),
    type = "button",
    class = "btn btn-default action-button",
    `data-val` = value,
    list(validateIcon(icon), label), ...
```

```
  )
}
```

The button tag has some attributes: id, style, type, class, `data-val`, label and children passed via `...`

When the app starts, the action button has the value 0 and each click will increment its value by 1. How is this behaviour created? For each Shiny input element (radio, slider), there is an associated JavaScript magic file, called input binding, which you can find here. In our case, we are only interested in the action button binding:

```
var actionButtonInputBinding = new InputBinding();
$.extend(actionButtonInputBinding, {
  find: function(scope) {
    return $(scope).find(".action-button");
  },
  getValue: function(el) {
    return $(el).data('val') || 0;
  },
  // ... other methods
});
```

What you see above is **not** the whole script since we focus on the first method, that is `find`. It will look for **all** elements having the class **.action-button**, making it possible to define multiple action buttons at the same time.

Consequently, if we go back to the previous section, the `actionButton` has the class **.action-button**, thereby making it visible to the binding. What is interesting is that all elements having the class .action-button will be considered by the same shiny input binding.

11.1.2 Application to tabler

First of all, let's compare the tabler HTML button to the Shiny action button.

```
<button class="btn btn-primary">Button</button>
```

We convert it to R. The button API contains more style and leave the reader to add extra elements as an exercise.

```
tabler_button <- function(inputId, label, status = NULL, icon = NULL, width = NULL, ..

  btn_cl <- paste0(
```

```
    "btn action-button",
    if (is.null(status)) {
      " btn-primary"
    } else {
      paste0(" btn-", status)
    }
  )

  value <- restoreInput(id = inputId, default = NULL)

  # custom right margin
  if (!is.null(icon)) icon$attribs$class <- paste0(
    icon$attribs$class, " mr-1"
  )

  tags$button(
    id = inputId,
    style = if (!is.null(width)) paste0("width: ", validateCssUnit(width), ";"),
    type = "button",
    class = btn_cl,
    `data-val` = value,
    list(icon, label), ...
  )
}
```

In tabler, the button status is mandatory, which is the reason why it is a parameter of the function. Moreover, we need to add an horizontal right margin to the icon, if provided so that the label renders well.

```
ui <- tabler_page(
  tabler_button("btn", "Click", icon = icon("thumbs-up"), width = "25%")
)

server <- function(input, output, session) {
  observe(print(input$btn))
}

shinyApp(ui, server)
```

We easily check that clicking on the button increments the related input. In few minutes, we were able to implement a custom tabler input button, built on top of the Shiny action button.

As a general rule, don't try to reinvent the wheel and see whether any existing Shiny element may be reused/adapted!

11.2 Toggle Switch

We implement the toggle switch component.

```
<label class="form-check form-switch">
  <input class="form-check-input" type="checkbox" checked>
  <span class="form-check-label">Option 1</span>
</label>
```

Notice that the tabler switch has the checkbox type, which is very similar to the Shiny checkbox (a switch is a checkbox with a different style)

```
checkboxInput("test", "Test", TRUE)
```

Test

Therefore, we should again be able to build on top of an existing input binding. We create the `tabler_switch` function:

```
tabler_switch <- function(inputId, label, value = FALSE, width = NULL) {

  value <- restoreInput(id = inputId, default = value)
  inputTag <- tags$input(
    id = inputId,
    type = "checkbox",
    class = "form-check-input"
  )

  if (!is.null(value) && value) {
    inputTag$attribs$checked <- "checked"
  }

  tags$label(
    class = "form-check form-switch",
    style = if (!is.null(width)) {
      paste0("width: ", validateCssUnit(width), ";")
    },
    inputTag,
    span(class = "form-check-label", label)
  )
}
```

Besides, we may also create an `update_tabler_switch` function similar to the `updateCheckboxInput`. We will also need `dropNulls`, a function that removes all NULL elements from a list (this function is often used in all custom Shiny

templates). If you remember the `sendInputMessage` from R will be received by the `receiveMessage` method on the JavaScript side.

```
dropNulls <- function (x) {
  x[!vapply(x, is.null, FUN.VALUE = logical(1))]
}

update_tabler_switch <- function (session, inputId, label = NULL, value = NULL) {
  message <- dropNulls(list(label = label, value = value))
  session$sendInputMessage(inputId, message)
}
```

In the following example, the action button toggles the switch input value when clicked.

```
ui <- tabler_page(
  tabler_switch("toggle", "Switch", value = TRUE, width = "25%"),
  tabler_button("update", "Go!")
)

server <- function(input, output, session) {
  observe(print(input$toggle))
  observeEvent(input$update, {
    update_tabler_switch(
      session,
      "toggle",
      value = !input$toggle
    )
  })
}

shinyApp(ui, server)
```

Et voilà! 2 inputs in few minutes.

11.3 Navbar menu input

We saw in Chapter 4 that it is quite easy to bind other elements than pure inputs (HTML elements with the input tag) to Shiny. As a reminder, we created a custom input binding to detect the state of a shinydashboard box (collapsed/uncollapsed). In chapter 10, we created the `tabler_navbar` as well as the `tabler_navbar_menu` and `tabler_navbar_menu_item`. As in shinydashboard, it would be nice to capture the currently selected tab to be able to perform actions on the server side, updating the selected tab based on a button click.

Where do we start? First of all, the first thing is to add an id attribute to the `tabler_navbar_menu` so that it holds the corresponding `input$id`. Whether to use `inputId` or `id` as a parameter name is up to you, but keep in mind that `inputId` does not exist in HTML.

```
tabler_navbar_menu <- function(..., inputId = NULL) {
  tags$ul(id = inputId, class = "nav nav-pills navbar-nav", ...)
}
```

The next step is the `navbarMenuBinding` creation. We decide to look for the `navbar-nav` class in the `find` method. Below, we describe the binding step by step. You may find the whole working code at the end of this example.

```
find: function(scope) {
  return $(scope).find('.navbar-nav');
}
```

In the `initialize` method, we ensure that if no tab is selected at start, the first tab will be by default. Otherwise, we select the activated tab. We use the string interpolation to ease the insertion of JS code in strings (`${menuId} .nav-link.active`).

```
initialize: function(el) {
  let menuId = '#' + $(el).attr('id');
  let activeTab = $(`${menuId} .nav-link.active`);
  // if multiple items are found
  if (activeTab.length > 0) {
    let tabId = $(activeTab).attr('data-value');
    $(activeTab).tab('show');
    $(`#${tabId}`).addClass('show active');
  } else {
    $(`${menuId} .nav-link`)
      .first()
      .tab('show');
  }
}
```

The role of `getValue` is to return the currently selected tab. As a reminder, here is the `tabler_navbar_menu_item` function:

```
tabler_navbar_menu_item <- function(text, tabName, icon = NULL, selected = FALSE) {

  item_cl <- paste0("nav-link", if(selected) " active")
```

```
  tags$li(
    class = "nav-item",
    a(
      class = item_cl,
      href = paste0("#", tabName),
      `data-toggle` = "pill", # see https://getbootstrap.com/docs/4.0/components/navs/
      `data-value` = tabName,
      role = "tab",
      span(class = "nav-link-icon d-md-none d-lg-inline-block", icon),
      span(class = "nav-link-title", text)
    )
  )
}
```

From that function, the active item has is the `a` element with the classes `nav-link active`. We recover the tab value stored in the `data-value` attribute. A bit of jQuery will do the trick!

```
getValue: function(el) {
  let activeTab = $(el).find('a').filter('nav-link active');
  return $(activeTab).attr('data-value');
}
```

`setValue` is the function allowing to update the active tab. Bootstrap 4 already has predefined methods to activate tabs. The easiest way is to select the tab by name like `$('#tabMenu a[href="#tab1"]').tab('show')`. The `receiveMessage` is simply applying the `setValue` method.

```
setValue: function(el, value) {
  let hrefVal = '#' + value;
  let menuId = $(el).attr('id');
  $(`#${menuId} a[href="${hrefVal}"]`).tab('show');
}

receiveMessage: function(el, data) {
  this.setValue(el, data);
}
```

Besides, we have to create the `update_tabler_tab_item` function.

```
update_tabler_tab_item <- function(inputId, value, session = getDefaultReactiveDomain()) {
  session$sendInputMessage(inputId, message = value)
}
```

`subscribe` will tell Shiny when to change the current input value and made it available in the whole app. We may listen to multiple events, keeping in mind that events occur in the following order:

- hide.bs.tab (on the current active tab)
- show.bs.tab (on the to-be-shown tab)
- hidden.bs.tab (on the previous active tab, the same one as for the hide.bs.tab event)
- shown.bs.tab (on the newly-active just-shown tab, the same one as for the show.bs.tab event)

Hence, it makes more sense to listen to `shown.bs.tab` (wait the current tab to be shown).

```
subscribe: function(el, callback) {
  // important to use shown.bs.tab and not show.bs.tab!
  $(el).on('shown.bs.tab.navbarMenuBinding', function(e) {
    callback();
  });
},

unsubscribe: function(el) {
  $(el).off('.navbarMenuBinding');
}
```

Below is a recap of the binding with the creation and registration included:

```
$(function() {
  // Input binding
  let navbarMenuBinding = new Shiny.InputBinding()
  $.extend(navbarMenuBinding, {
    find: function(scope) {
      return $(scope).find('.navbar-nav');
    },
    initialize: function(el) {
      let menuId = '#' + $(el).attr('id');
      let activeTab = $(`${menuId} .nav-link.active`);
      // if multiple items are found
      if (activeTab.length > 0) {
        let tabId = $(activeTab).attr('data-value');
        $(activeTab).tab('show');
        $(`#${tabId}`).addClass('show active');
      } else {
        $(`${menuId} .nav-link`)
          .first()
```

```
          .tab('show');
      }
    },
    // Given the DOM element for the input, return the value
    getValue: function(el) {
      let activeTab = $(el).find('a').filter('nav-link active');
      return $(activeTab).attr('data-value');
    },
    setValue: function(el, value) {
      let hrefVal = '#' + value;
      let menuId = $(el).attr('id');
      $(`#${menuId} a[href="${hrefVal}"]`).tab('show');
    },
    receiveMessage: function(el, data) {
      this.setValue(el, data);
    },
    subscribe: function(el, callback) {
      $(el).on('shown.bs.tab.navbarMenuBinding', function(event) {
        callback();
      });
    },
    unsubscribe: function(el) {
      $(el).off('.navbarMenuBinding');
    }
  });

  Shiny.inputBindings.register(navbarMenuBinding, 'text');
});
```

We test the new `navbar_menu` binding below.

```
ui <- tabler_page(
  tags$head(
    tags$script(
      HTML(
        "$(function() {
          // Input binding
          let navbarMenuBinding = new Shiny.InputBinding()
          $.extend(navbarMenuBinding, {
            find: function(scope) {
              return $(scope).find('.navbar-nav');
            },
            initialize: function(el) {
              let menuId = '#' + $(el).attr('id');
              let activeTab = $(`${menuId} .nav-link.active`);
```

```
            // if multiple items are found
            if (activeTab.length > 0) {
              let tabId = $(activeTab).attr('data-value');
              $(activeTab).tab('show');
              $(`#${tabId}`).addClass('show active');
            } else {
              $(`${menuId} .nav-link`)
                .first()
                .tab('show');
            }
          },
          // Given the DOM element for the input, return the value
          getValue: function(el) {
            let currentTab = $(el).find('a').filter('.nav-link.active');
            return $(currentTab).attr('data-value');
          },
          setValue: function(el, value) {
            let hrefVal = '#' + value;
            let menuId = $(el).attr('id');
            $(`#${menuId} a[href=\"${hrefVal}\"]`).tab('show');
          },
          receiveMessage: function(el, data) {
            this.setValue(el, data);
          },
          subscribe: function(el, callback) {
            $(el).on('shown.bs.tab.navbarMenuBinding', function(event) {
              callback();
            });
          },
          unsubscribe: function(el) {
            $(el).off('.navbarMenuBinding');
          }
        });

        Shiny.inputBindings.register(navbarMenuBinding, 'text');
      });
      "
    )
  )
),
tabler_navbar(
  brand_url = "https://preview-dev.tabler.io",
  brand_image = "https://preview-dev.tabler.io/static/logo.svg",
  nav_menu = tabler_navbar_menu(
    id = "current_tab",
```

```
      tabler_navbar_menu_item(
        text = "Tab 1",
        icon = NULL,
        tabName = "tab1",
        selected = TRUE
      ),
      tabler_navbar_menu_item(
        text = "Tab 2",
        icon = NULL,
        tabName = "tab2"
      )
    ),
    tabler_button("update", "Change tab", icon = icon("exchange-alt"))
  ),
  tabler_body(
    tabler_tab_items(
      tabler_tab_item(
        tabName = "tab1",
        sliderInput(
          "obs",
          "Number of observations:",
          min = 0,
          max = 1000,
          value = 500
        ),
        plotOutput("distPlot")
      ),
      tabler_tab_item(
        tabName = "tab2",
        p("Second Tab")
      )
    ),
    footer = tabler_footer(
      left = "Rstats, 2020",
      right = a(href = "https://www.google.com")
    )
  )
)
server <- function(input, output, session) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs))
  })

  observeEvent(input$current_tab, {
    showNotification(
```

```
      paste("Hello", input$current_tab),
      type = "message",
      duration = 1
    )
  })

  observeEvent(input$update, {
    newTab <- if (input$current_tab == "tab1") "tab2" else "tab1"
    update_tabler_tab_item("current_tab", newTab)
  })
}
shinyApp(ui, server)
```

Here we are!

Chapter 12

Adding more interactivity

In this part, we are going to add more life to the template element. We first see how to enhance an existing static HTML component. Then we will explore complex feedback mechanisms to provide more interactivity to your app.

12.1 Custom progress bars

Progress bars are a good way to display metric related to a progress, for instance tracking the number of remaining tasks for a project. In general, those elements are static HTML. Hence, it would be interesting to be able to update the current value from the server side. Since it is not an proper input element, implementing an input binding is inapropriate and we decide to proceed with the `sendCustomMessage`/`addCustomMessageHandler` pair. We first create the `tabler_progress` tag which is mainly composed of:

- `style` gives the current progress value. This is the main element
- `min` and `max` are bounds, in general between 0 and 100
- `id` ensures the progress bar uniqueness, thereby avoiding conflicts

```
tabler_progress <- function(id = NULL, value) {
  div(
    class = "progress",
    div(
      id = id,
      class = "progress-bar",
      style = paste0("width: ", value, "%"),
      role = "progressbar",
      `aria-valuenow` = as.character(value),
```

```
      `aria-valuemin` = "0",
      `aria-valuemax` = "100",
      span(class = "sr-only", "38% Complete")
    )
  )
}

update_tabler_progress <- function(id, value, session = shiny::getDefaultReactiveDomai
  message <- list(id = session$ns(id), value = value)
  session$sendCustomMessage(type = "update-progress", message)
}
```

The next element is the `update_tabler_progress` function which send 2 elements from R to JS:

- The progress `id`
- The new value

On the JS side, we have a basic `addCustomMessageHandler`. As mentionned in Chapter 3, `sendCustomMessage` and `addCustomMessageHandler` are connected by the `type` parameter. This is crucial! Moreover, as the sent message is a R list, it becomes an JSON, meaning that elements must be accessed with a `.` in JS:

```
$(function () {
  Shiny.addCustomMessageHandler('update-progress', function(message) {
    $('#' + message.id).css('width', message.value +'%');
  });
});
```

We finally test these components in a simple app:

```
ui <- tabler_page(
  tags$head(
    tags$script(
      "$(function() {
        Shiny.addCustomMessageHandler('update-progress', function(message) {
          $('#' + message.id).css('width', message.value +'%');
        });
      });
      "
    )
  ),
  tabler_body(
```

```
    sliderInput(
      "progress_value",
      "Progress value:",
      min = 0,
      max = 100,
      value = 50
    ),
    tabler_progress(id = "progress1", 12)
  )
)

server <- function(input, output, session) {
  observeEvent(input$progress_value, {
    update_tabler_progress(
      id = "progress1",
      input$progress_value
    )
  })
}
shinyApp(ui, server)
```

NOTE: How to handle custom messages in shiny modules? Well, it is pretty straightforward: we wrap any id with the module namespace given by `session$ns()` before sending it to JS. You may even do it by default (without modules) like in the previous example since `session$ns()` will be `""`.

12.2 User feedback: toasts

Toasts are components to send discrete user feedback, contrary to modals which open in the middle of the page. Toasts may open on all sides of the window and are similar to the Shiny notifications (see here). The tabler toast component is built on top of Bootstrap 4. Therefore, we will rely on this documentation.

12.2.1 Toast skeleton

The skeleton is the HTML structure of the toast:

```
<div class="toast show" role="alert" aria-live="assertive" aria-atomic="true" data-autohide="fals
  <div class="toast-header">
    <span class="avatar mr-2" style="background-image: url(...)"></span>
    <strong class="mr-auto">Mallory Hulme</strong>
    <small>11 mins ago</small>
```

```
    <button type="button" class="ml-2 close" data-dismiss="toast" aria-label="Close">
      <span aria-hidden="true">&times;</span>
    </button>
  </div>
  <div class="toast-body">
    Hello, world! This is a toast message.
  </div>
</div>
```

Toasts are mainly composed of a header and a body. There might be a close button in case the toast does not hide itself. If multiple toasts appear one after each others, they are stacked, the latest being at the bottom of the stack. The position is controled with the style attribute like `style="position: absolute; top: 0; right: 0;"` for a top-right placement. Accessibility parameters like `aria-live` are detailed here.

12.2.2 The toast API

Toasts have a JS API to control their behaviour, `$('<toast_selector>').toast(option)`, where option stend for:

- animation applies a CSS fade transition to the toast and is TRUE by default
- autohide automatically hides the toast (TRUE by default)
- delay is the delay to hide the toast (500 ms)

There are 3 methods: hide, show and dispose (dispose ensures the toast does not appear anymore). Finally, we may fine tune the toast behavior with 4 events: show.bs.toast, shown.bs.toast, hide.bs.toast, hide.bs.toast, hidden.bs.toast (like for tabs).

12.2.3 R implementation

We first create the toast skeleton. We assume our toast will hide automatically, so that we may remove the delete button as well as the `data-autohide="false` attribute. All parameters are optional except the toast id, which is required to toggle the toast:

```
tabler_toast <- function(id, title = NULL, subtitle = NULL, ..., img = NULL) {
  div(
    id = id,
    class = "toast",
    role = "alert",
```

```
    style = "position: absolute; top: 0; right: 0;",
    `aria-live` = "assertive",
    `aria-atomic` = "true",
    `data-toggle` = "toast",
    div(
      class = "toast-header",
      if (!is.null(img)) {
        span(
          class = "avatar mr-2",
          style = sprintf("background-image: url(%s)", img)
        )
      },
      if (!is.null(title)) strong(class = "mr-auto", title),
      if (!is.null(subtitle)) tags$small(subtitle)
    ),
    div(class = "toast-body", ...)
  )
}
```

We create the `show_tabler_toast` function. Since the toast automatically hides, it does not make sense to create the hide function, as well as the dispose.

```
show_tabler_toast <- function(id, options = NULL, session = getDefaultReactiveDomain()) {
  message <- shiny:::dropNulls(
    list(
      id = id,
      options = options
    )
  )
  session$sendCustomMessage(type = "tabler-toast", message)
}
```

The corresponding JS handler is given by:

```
$(function() {
  Shiny.addCustomMessageHandler('tabler-toast', function(message) {
    $(`#${message.id}`)
      .toast(message.options)
      .toast('show');

    // add custom Shiny input to listen to the toast state
    $(`#${message.id}`).on('hidden.bs.toast', function() {
      Shiny.setInputValue(message.id, true, {priority: 'event'});
    });
```

```
  });
});
```

We first configure the toast and ask him to show. Notice how we chained jQuery methods! We optionally add an event listener to capture the `hidden.bs.toast` event, so that we may trigger an action when the toast is closed. The `input$id` will be used for that purpose in combination with the `Shiny.setInputValue`. Notice the extra parameter `{priority: 'event'}`: basically, once the toast is closed, `input$id` will always be TRUE, thereby breaking the reactivity. Adding this extra parameter forces the evaluation of the input, although constant over time.

12.2.4 Wrap up

```
ui <- tabler_page(
  tags$head(
    tags$script(
      "$(function() {
        Shiny.addCustomMessageHandler('tabler-toast', function(message) {
          $(`#${message.id}`)
            .toast(message.options)
            .toast('show');

          // add custom Shiny input to listen to the toast state
          $(`#${message.id}`).on('hidden.bs.toast', function() {
            Shiny.setInputValue(message.id, true, {priority: 'event'});
          })
        });
      });
      "
    )
  ),
  tabler_toast(
    id = "toast",
    title = "Hello",
    subtitle = "now",
    "Toast body",
    img = "https://preview-dev.tabler.io/static/logo.svg"
  ),
  tabler_button("launch", "Go!", width = "25%")
)

server <- function(input, output, session) {
```

```
  observe(print(input$toast))
  observeEvent(input$launch, {
    show_tabler_toast(
      "toast",
      options = list(
        animation = FALSE,
        delay = 2000
      )
    )
  })

  observeEvent(input$toast, {
    showNotification(
      "Toast was closed",
      type = "warning",
      duration = 1,

    )
  })
}

shinyApp(ui, server)
```

12.3 Transform an element in a custom action button

As seen in Chapter 11, any `<button>`, `<a>` element holding the `action-button` class may eventually become an action button. The tabler template has dropdown menus in the navbar and we would like to transform dropdown items in action buttons. The `tabler_dropdown` functions takes the following parameters:

- id is required by the `show_tabler_dropdown` (see below) function which opens the menu
- title is the dropdown menu name
- subtitle is optional text
- img is an optional image
- … hosts the `tabler_dropdown_item` (see below)

```
tabler_dropdown <- function(..., id = NULL, title, subtitle = NULL, img = NULL) {

  div(
```

```
    class = "nav-item dropdown",
    a(
      href = "#",
      id = id,
      class = "nav-link d-flex lh-1 text-reset p-0",
      `data-toggle` = "dropdown",
      `aria-expanded` = "false",
      if (!is.null(img)) {
        span(class = "avatar", style = sprintf("background-image: url(%s)", img))
      },
      div(
        class = "d-none d-xl-block pl-2",
        div(title),
        if (!is.null(subtitle)) {
          div(class = "mt-1 small text-muted", subtitle)
        }
      )
    ),
    div(class = "dropdown-menu dropdown-menu-right", `aria-labelledby` = id, ...)
  )
}
```

To convert a dropdown item in an action button , we add the `action-button` class as well as the `id` parameter to recover the corresponding input id.

```
tabler_dropdown_item <- function(..., id = NULL) {
  a(id = id, class = "dropdown-item action-button", href = "#", ...)
}
```

We finally create the `show_tabler_dropdown` as well as the corresponding Shiny message handler.

```
show_tabler_dropdown <- function(id, session = getDefaultReactiveDomain()) {
  session$sendCustomMessage(type = "show-dropdown", message = id)
}
```

To show the dropdown, we use the `dropdown` method which is linked to the `data-toggle="dropdown"` of `tabler_dropdown`.

```
$(function() {
  Shiny.addCustomMessageHandler('show-dropdown', function(message) {
    $(`#${message}`).dropdown('show');
  });
});
```

Let's play with it!

```
ui <- tabler_page(
  tags$head(
    tags$script(
      "$(function() {
        Shiny.addCustomMessageHandler('show-dropdown', function(message) {
          $(`#${message}`).dropdown('show');
        });
      });
      "
    )
  ),
  tabler_navbar(
    brand_url = "https://preview-dev.tabler.io",
    brand_image = "https://preview-dev.tabler.io/static/logo.svg",
    nav_menu = NULL,
    tabler_dropdown(
      id = "mydropdown",
      title = "Dropdown",
      subtitle = "click me",
      tabler_dropdown_item(
        id = "item1",
        "Show Notification"
      ),
      tabler_dropdown_item(
        "Do nothing"
      )
    )
  ),
  tabler_body(
    tabler_button("show", "Open dropdown", width = "25%"),
    footer = tabler_footer(
      left = "Rstats, 2020",
      right = a(href = "https://www.google.com")
    )
  )
)
server <- function(input, output, session) {

  observeEvent(input$show, {
    show_tabler_dropdown("mydropdown")
  })

  observeEvent(input$item1, {
    showNotification(
```

```
      "Success",
      type = "message",
      duration = 2,

    )
  })
}
shinyApp(ui, server)
```

12.4 Tab events

Do you remember about the navbar element and the tabsetpanel system of Chapter 10? Navs allow to organize any app into several tabs, acting like pages. This is a powerful tool for Shiny since it is currently not straightforward to create multi-pages Shiny apps like anyone would do with a website. Navs relie on the Bootstrap4 API but we only used few JS functions.

12.4.1 Insert/Remove tabs in tabsetpanel

How about dynamically inserting/removing tabs from a `tabler_navbar`? I chose this example since it involves extra technical details about Shiny.

TO DO: start with a naive approach, show it does not work and introduce `Shiny.renderContent`, `processDeps` ...

12.4.2 More events

There exists events that trigger after/before showing/hiding a tabs, that is `hidden.bs.tab` and `shown.bs.tab` (`hide.bs.tab` and `show.bs.tab` are triggered before). For instance, we may recover the previously selected tab, namely `e.relatedTarget`, and store it in a shiny input with `Shiny.setInputValue`:

```
$(function() {
  $('a[data-toggle="tab"]').on('shown.bs.tab', function (e) {
    Shiny.setInputValue('previous_tab', e.relatedTarget)
  });
});
```

TO FINISH

Chapter 13

Testing templates elements

Beautify with CSS and SASS

This part will introduce you to tools like {fresh} and {bootstraplib}

Chapter 14

Beautify with fresh