For many of us, getting meaningful data for our analyses is one of the more difficult parts of our jobs and one that does not always appear automatable. As someone who strives to elevate myself and my teams away from the production of data sets, this is particularly frustrating. Luckily, many companies provide APIs to make it easier to fetch the data you need and there are great tools such as Galvanize Robots that makes it even easier to do this, with connectors for a wide variety of systems, both cloud and on premise.

Sadly, this is not the case with all providers and so we are left to our manual efforts to fetch what we need, especially when the provider generates downloadable reports. Using three tools we can automate away the entire data-fetch process. Fair warning, there is some setup required first, but its worth it. Be careful which sites you scrape (don’t hammer small guys sites for example), review their fair use policies and be a good citizen.

Lets get started.

**Setup**

Download and install the following:

1. R
2. R-studio (technically you don’t have to, but I would recommend it)
3. Docker

I’m not going to go over the installation here as I’m going to assume you know how to google any problems you have or can ask your IT department for help.

**Setting up Docker**

We are going to be using Docker to spin up a Linux environment on which we will run an application called Selenium. Selenium is a web-testing tool that allows you to automate the clicking of buttons, amongst other things, which is perfect for our data-fetching problem.

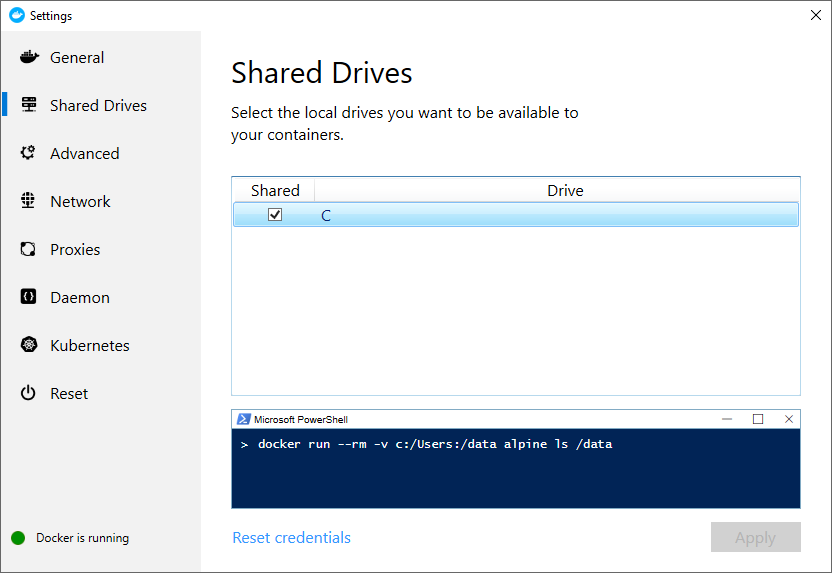
Before we get that far however, we need to give docker permission to write to a location on one’s computer. I run a Windows machine, so if you use Linux or a Mac, you will have to do your own digging on that.

In the systems tray, look for a little whale with containers on it. . If you don’t see one, it means that docker is not running on your machine; go to the start menu and run the application. It will take a moment to start, so be patient.

Right click and go to Settings and then to Shared Drives. Check the checkbox so the drive you want to

Search Google or type a URL

use is shared. It should look like this:

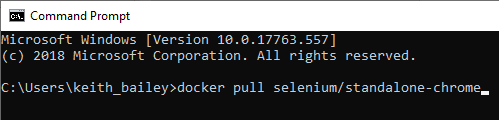


Depending on what access rights you have on your machine, you may need IT to help you with this.

If we want to have a Selenium container that will run commands for us, we need an image from which it can be run. Luckily, there is the docker image repository and with a simple command we can grab the latest version of a selenium image.

From the start menu, run cmd to get the command prompt up. Then issue the following command:

*docker pull selenium/standalone-chrome*



**Setting up the script and populating your clicks**

Go to this github page and download my code and clicks1 file.

Run RStudio from the start menu and open the click\_automation.R file.

Before we run anything here, we need to change a couple of variables for your own setup; clicks and destination. Clicks needs to be set to the full file path of the click1.xlsx file you just downloaded and destination is where you want the downloaded files to be copied to.

Everything is now setup to download the Microsoft financial data for you. If you just want to run this first without reading on, hit the ‘Source’ button in Rstudio.

The first time you run the script will be slower as we have some base packages that need to be installed, that do much of the heavy lifting.

It takes a bit of time to run each command. As a rule of thumb it takes about 10 seconds to go through each page (to make sure the page has finished loading), plus the start up and shut down of the Selenium server.

**Setting up the Clicks File**

Starting with a simple example open the clicks1.xlsx. In the clicks excel file I have populated some examples, so you can see where the data needs to go.

|  |  |
| --- | --- |
| Field | Explanation |
| step | Free text field to explain what we are doing |
| sendKeysToElement | Used to send data to a page, such as login information |
| perform\_download | TRUE/FALSE – if set to false it will not try to copy the file to your local machine. |
| append\_to\_name | Use alphanumeric only - Appends this to the name of the file that was downloaded. This is particularly useful for sites that create meaningless filenames, such as ID numbers. E.g. first row in clicks1.xlsx “MSFT Income Statement.csv” would become “MSFT Income StatementMarketData2019-08-02\_11.51.28.csv” |
| final\_click | The css selector that we want to run last on this page |
| css\_button | JSON format - Allows for multiple elements to be clicked BEFORE the export button is triggered |
| urls | Full url including http:// - the page we want to navigate to |

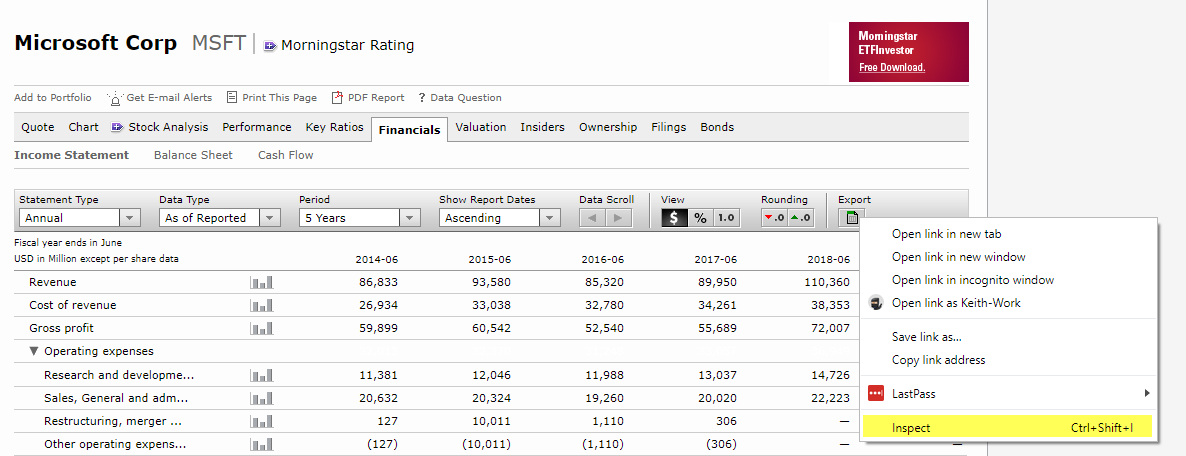
Essentially the page in the URL will be loaded and all the buttons in css\_button cell will be clicked (if any data present) with the data in the final\_click cell always run last. To get a proper understanding of what is happening, let’s walk through an example of Microsoft data from Morningstar.

First, navigate to [https://www.morningstar.ca](https://www.morningstar.ca/ca/?culture=en-CA); in the stock section, search for MSFT, which should take you to Quote information about the stock. As I want the financial information, click through to Financials then the All Financial Data link beneath the main table. This will open a new window where we can export the data as a CSV.

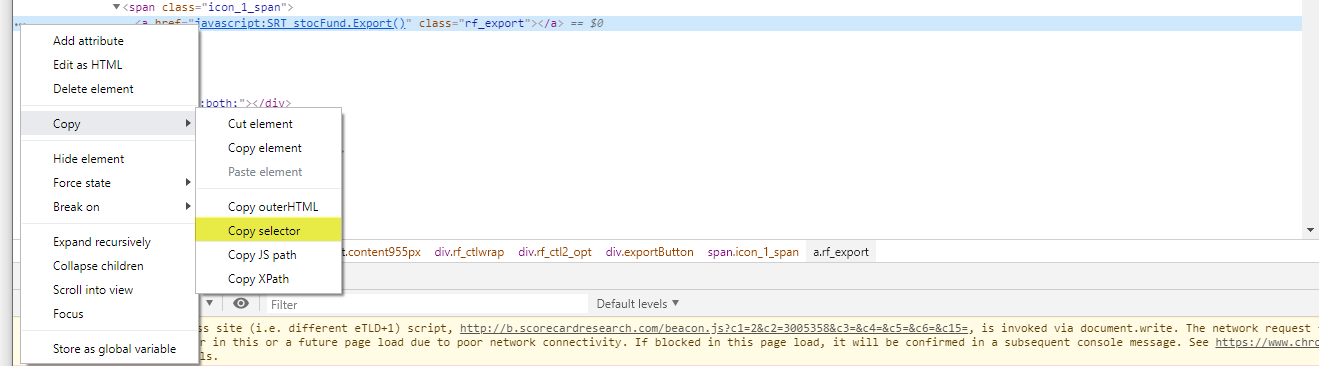
What we need from this page is the instructions we want to send to Selenium so that it clicks the things we would manually click.

**Getting the click information**

Right click on the button, or link, you want to be clicked. In the dialog box that opens, click inspect.



In the new window, a line will be highlighted to the left of which will be three dots. Click on those dots and got to Copy 🡪 Copy Selector.



It is this piece of information we need to paste into the final\_click cell of the clicks1 speadsheet as that is what will be sent to Selenium to be performed. You would need to do this for each URL you want to pull data from. In the click1.xlsx example, there are 3 lines for Income Statement, Balance Sheet and Ratios each of which come from different web pages (URL).

Now we have it setup, we can run the code.

**More Complicated things**

**Clicking things before we download data**

This time, we are going to use the clicks2 file, so make sure you change the ‘clicks’ variable to the location of the clicks2.xlsx file.

With the Morningstar data, I noticed that it is rounded to thousands of dollars. I would rather have it without rounding. First, I want to click on the round down button BEFORE I click export. The way I have setup this tool is using JSON so that we could issue more than one command if we need to. In this case, we only need one command before we click the export button.

The command doesn’t look nice, so I will split it down.

{"click\_data":[

{

"click1":

"#sfcontent > div.rf\_ctlwrap > div.rf\_ctl2\_opt > div.roundingButton > span > a.rf\_rounddn"

}

]}

If you wanted to add more clicks, you would need to add more here like this:

{"1":[

{

"click1":

"#sfcontent > div.rf\_ctlwrap > div.rf\_ctl2\_opt > div.roundingButton > span > a.rf\_rounddn"

},

{

"click2":

"#sfcontent > div.rf\_ctlwrap > div.rf\_ctl2\_opt > div.roundingButton > span > a.rf\_rounddn"

},

]}

If we used the code above, it would click the round down button twice. If, as is the case with Morningstar, the button disappears when we can no longer round down and we STILL ask Selenium to click the button, it will try, fail, log this information for you and move on. It will not stop the script entirely.

**Sending data to the page**

Often, we will want to grab data from sites that need input first. A classic example would be username password, but you will also run into things like dates or drop-down boxes.

In the clicks file, I created an example login\_step you can use

The sendKeysToElement field lets us do just that. Let’s use the <https://www.morningstar.com/sign-in> page as example here. We need to provide our email address and password.

Even though we want to grab the ID of the element, we can do the same thing as before. Right click, Inspect, three dots, copy selector. It will nicely just grab the ID for us.

e.g.

Email - #mdc-text-field\_\_UID\_\_1002

Password - #mdc-text-field\_\_UID\_\_1006

Notice, however, that we want to drop the # from the beginning.

{"1":[{"using":"id","value":" mdc-text-field\_\_UID\_\_1002","sendKey":"name@youremail.com"}],"2":[{"using":"id","value":"mdc-text-field\_\_UID\_\_1006","sendKey":"your\_password"}]}

Once we have populated our login data, we need to click the login button. Just as before, grab the css selector of the login button element. In this example that would be:

#\_\_layout > div > div.mdc-page-shell\_\_content.mds-page-shell\_\_content > main > div > form > div.mdc-sign-in-form\_\_actions > button

**Screenshot**

One of the many useful features of Selenium is the ability to get a screenshot of the page that is loaded. This can be at anytime, before or after we have sent data to the page or clicked any buttons. This is helpful as it lets us see what the page looks like on the docker container. I have used this when I ran into errors. The default in my code is for this to be enabled; should you not want this you can disable this in the code by setting screenshotLogging = FALSE.

Unless you are familiar with R and have changed your working directory, these screenshot images should be saved to your Documents folder.