Migration from Excel to R: Getting Started

Every migration starts with proper data preparation. This step might take some time if a company has a lot of Excel workbooks. The most important thing is to extract original data from analyzed data, create tables, and save them in a **CSV format**. If you're just starting – start small. Pick one or two CSVs that move the needle for a start. In almost any use case, you don't have to start with a complete dataset. Remember:

- When dealing with multiple sheets in a workbook, you need to combine them into one or divide them into different CSVs.
- When combining all sheets from the workbook into one table, make sure all sheets have the same number of columns and the same column names.

It would make sense to switch from CSV to SQL in the future, but using CSV is not a dealbreaker in the beginning. Moreover, you've likely performed some useful and effective analytical operations within Excel. Don't get rid of them just yet. By all means, analyze/verify these operations one more time, and describe their logic in detail (to recreate it later in R).

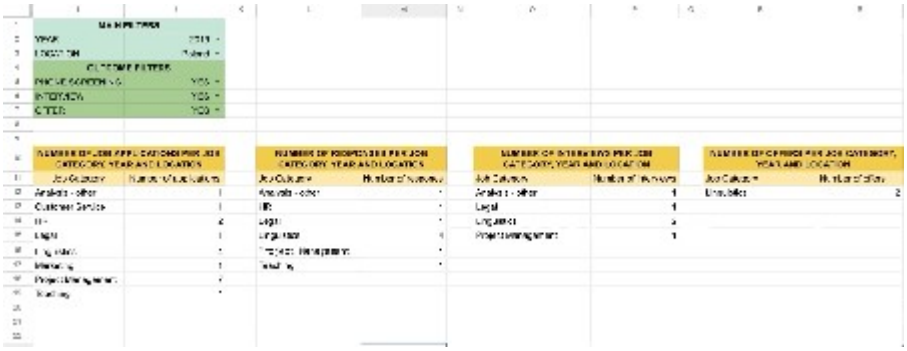
Sample Case: Job Hunt Analysis

Last year, one of our consultants was searching for a full-time job, and they started tracking application processes to see a more accurate picture of their career prospects – primarily to find out which companies/industries find a single profile interesting. Information about all applications sent in 2019 and 2020 were stored in two Excel sheets (original data) and one in Google Sheets (analysis). Let's take a quick preview of this data.

Original data:

	A	B	C	D	E	F	G
1	YEAR	COUNTRY	JOB CATEGORY	PHONE SCREENING	INTERVIEW	OFFER	IN THE FUTURE
2	2019	United States	Linguistics	NO	NO	NO	NO
3	2019	United States	Linguistics	YES	NO	NO	NO
4	2019	United Kingdom	Linguistics	YES	YES	NO	YES
5	2019	Poland	Linguistics	YES	YES	YES	YES
6	2019	United Kingdom	Linguistics	NO	NO	NO	NO
7	2019	United States	Linguistics	NO	NO	NO	NO
8	2019	United Kingdom	Linguistics	YES	NO	NO	NO
9	2019	United Kingdom	Linguistics	YES	YES	NO	YES
10	2019	Poland	Project Management	YES	YES	NO	YES
11	2019	United Kingdom	Project Management	NO	NO	NO	NO
12	2019	Poland	Customer Service	NO	NO	NO	NO
13	2019	Poland	Linguistics	YES	YES	YES	YES
14	2019	China	Linguistics	NO	NO	NO	NO
15	2019	Poland	Analysis - other	YES	YES	NO	YES
16	2019	Spain	Linguistics	NO	NO	NO	NO
17	2019	United States	Linguistics	NO	NO	NO	NO
18	2019	Ireland	Linguistics	YES	YES	NO	NO
19	2019	Poland	Marketing	NO	NO	NO	NO
20	2019	Poland	Project Management	NO	NO	NO	NO
21	2019	Poland	Project Management	NO	NO	NO	NO
22	2019	Poland	Project Management	NO	NO	NO	NO
23	2019	Poland	Project Management	NO	NO	NO	NO
24	2019	Poland	HR	YES	NO	NO	NO

Analysis:



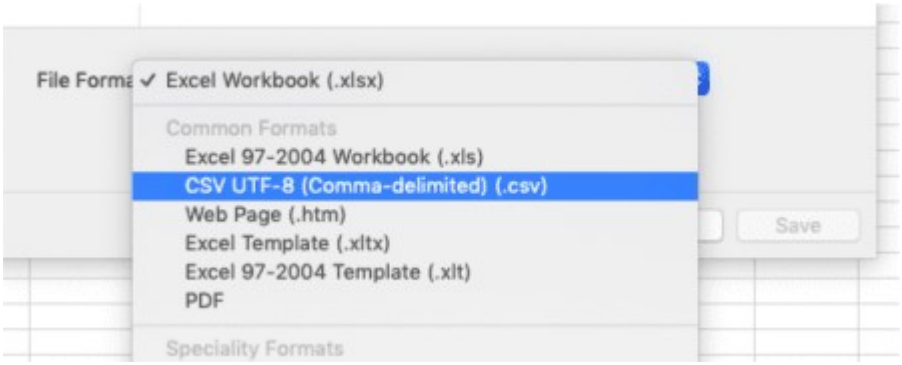
As you can see in the screen recording above, the dashboard works fine. However, there are some issues we encountered while using and maintaining it:

- When sharing the analysis with others, it is impossible to filter data without granting editing rights to users.
- We can't select multiple options in filters, e.g. we are not able to check the results for Poland and Spain simultaneously.
- We need to maintain four different tables and functions that show almost the same thing.

Because of the reasons above, it makes sense to migrate this dashboard to R Shiny and see if these problems can be eliminated.

Data Preparation

In this case, data preparation was fairly easy. Two sheets were merged into one, some values got replaced (YES -> 1, NO -> empty cell) using *Find and Replace*, and the data was saved to a CSV file:

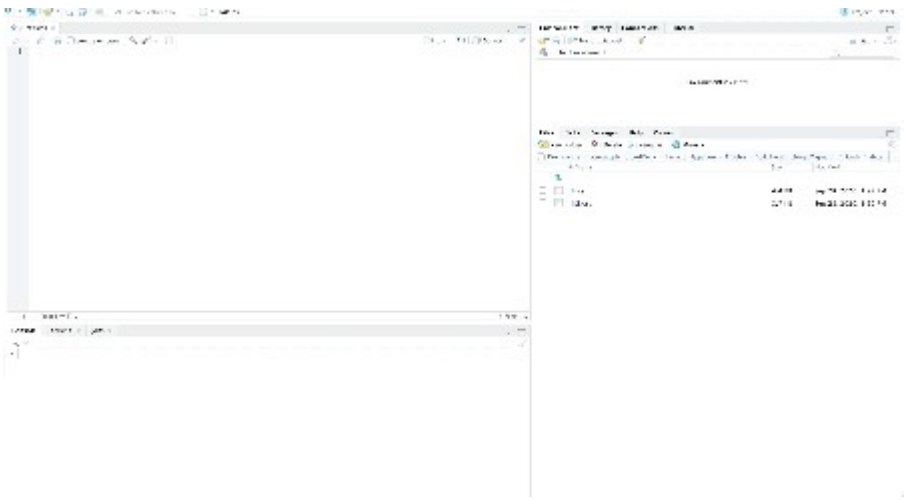


The next logical step is describing the features we need to migrate from Excel to R Shiny. The following table summarizes the steps pretty well:

Type	Live dashboard (yellow tables update each time new record is added, changed, or deleted)
Data to display as tables	Total view: number of applications submitted per job category (only MAIN FILTERS) • Number of positive replies per job category (MAIN + respective OUTCOME FILTER) • Number of interviews per job category (MAIN + respective OUTCOME FILTER) • Number of offers per job category (MAIN + respective OUTCOME FILTER)
Features to migrate	Filters (per each column), data aggregation (per job category)
Migration wishes	Multi-select in filtering, one table instead of four

Loading a Dataset

To start the migration process, we downloaded RStudio (a free development environment for R from our partner RStudio, PBC), found the CSV file we wanted to use, and imported it into RStudio:



After a successful import, the file appeared in the Global Environment. R had no problem with recognizing the CSV table.

> New to R or want to speed up your workflow? Check out our favorite RStudio Shortcuts and Tricks.

Creating Your First R Shiny Dashboard

Let's start simple with something that remotely resembles the original dashboard. The main goal is to make a simple app that displays the source data and filters it by *Job Category*.

It's important to understand two main components of an R Shiny app – the *UI* (User Interface) and the *server*. UI is a graphic layout of an app – everything the user sees on a webpage. The server is the backend of an application. The app is stored on the computer that runs R in the form of a page that can be viewed in a web browser.

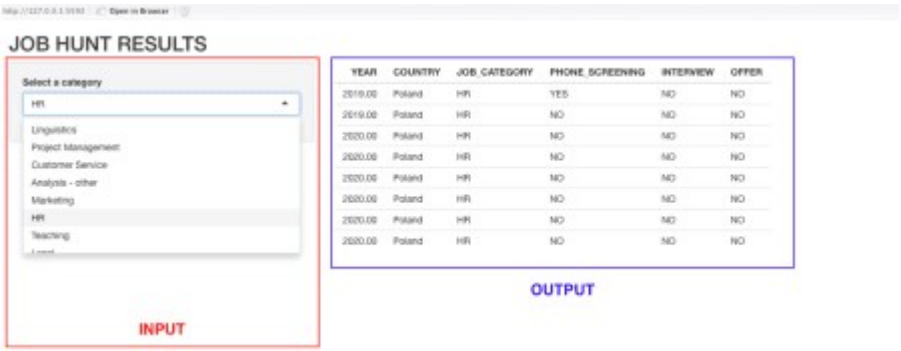
> If you are a beginner with R Shiny, here's an additional resource to help you get started: (Video Tutorial) Create and Customize a Simple Shiny Dashboard

Note: To share the R Shiny app with others, you either need to send them a copy of a script or host this page via an external web server.

To start, let's use the most basic Shiny app template:

```
library(shiny)
ui <- fluidPage()
server <- function(input, output) {}
shinyApp(ui=ui, server=server)
```

Defining Input and Output



Input is everything the user can interact with on a website. To name a few:

- select boxes – ***selectInput()***
- radio buttons – ***radioButtons()***
- sliders – ***sliderInput()***
- date ranges – ***dateRangeInput()***
- passwords – ***passwordInput()***

Each input must have an ***inputId*** (local name, e.g. 'value'), and a ***label*** (a description that will be displayed in an app, e.g. 'Select value'). In addition, depending on the type of input, you can provide additional parameters that will specify/limit the actions a user can perform. For more on defining ***input*** and ***output***, and other aspects of Shiny, read this tutorial by RStudio.

In the first draft of the app, let's create a reactive select box from which the user can choose any job category that appears in the dataset. Therefore, besides defining ***inputId*** and a label we need a list of choices for a dropdown list (***choices = TableName$ColumnName***):

```
selectInput('jobcategory', 'Select a category', choices =
j_h$JOB_CATEGORY)
```

output is the second argument in ***fluidPage()***. In this case, it is the result of actions taken by the user in inputs. It can be displayed in the form of a graph – ***plotOutput()***, table – ***tableOutput()***, text – ***textOutput()***, image – ***imageOutput()***, etc. Just like input, output needs to have an ID – ***outputId***. We'll display the results as a table, so let's use the ***tableOutput()*** function and name our reactive output 'jobhuntData':

```
tableOutput('jobhuntData')
```

Like many basic Shiny apps, our draft Shiny app is quite ugly by default. Let's fix this with some elements: ***titlePanel()***, ***sidebarLayout()***, ***sidebarPanel()***, and ***mainPanel()***.

At this point, after adding all elements to a ***fluidPage()*** function, our code looks like this:

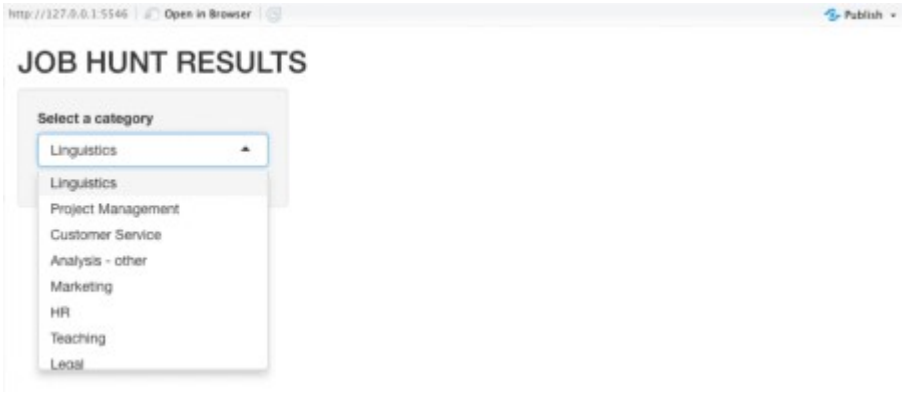
```
library(shiny)

ui <- fluidPage(
    titlePanel('JOB HUNT RESULTS'),
    sidebarLayout(
        sidebarPanel(
            selectInput('jobcategory', 'Select a category',
choices=j_h$JOB_CATEGORY)
        ),
        mainPanel(
            tableOutput('jobhuntData')
```

```
        )
    )
)

server <- function(input, output) {}

shinyApp(ui=ui, server=server)
```



We can see the filter, but there is no table yet. This is because R Shiny does not know what kind of table we want to generate. Let's introduce server requirements to address this.

> Want to Make a Beautiful Shiny App Fast? Use Appsilon's shiny.semantic open source package, which brings the Fomantic UI library to Shiny for attractive UI and rapid development.

How to Use Shiny Server

To build the first draft of the app, we need to create a source for the ***tableOutput()*** function by using a Render Function. Render Functions (e.g. ***renderImage()*** to render an image, ***renderPlot()*** to render a plot/graph, ***renderText()*** to render text, etc.) turn an R object into HTML, and place it in a Shiny webpage.

Below you can see how we assigned the ***outputId*** ("jobhuntData") to a function that renders the desired output – in our case, ***renderTable()*** to render a table. Inside this function, we specified data that we want to see in the table. Please mind that ***input$jobcategory*** refers to the Input Function from the UI, and it is always equal to the current value of the input (a value selected by a user).

```
library(shiny)

ui <- fluidPage(
    titlePanel('JOB HUNT RESULTS'),
    sidebarLayout(
        sidebarPanel(
            selectInput('jobcategory', 'Select a category',
choices=j_h2$JOB_CATEGORY)
        ),
        mainPanel(
            tableOutput('jobhuntData')
        )
    )
)
```

```
server <- function(input, output) {
    output$jobhuntData <- renderTable({
        jobcategoryFilter <- subset(j_h2, j_h2$JOB_CATEGORY ==
input$jobcategory)
    })
}

shinyApp(ui=ui, server=server)
```

The current version of the app does not look amazing, but we can see that the correct data is shown, and the server generates proper output according to the input provided by the user:

Migration – SQL and ShinyWidgets

Now that we know how to create a basic dashboard in R Shiny, we are going to migrate other features from our original dashboard. First and foremost, we had to not only create filters for all columns but also aggregate/group data by *YEAR* and *COUNTRY*. There are several ways to modify the dataset in R, but we decided to do it using an *SQL SELECT* statement. SQL is another topic on its own, but we recommend that you learn the basics of SQL if you work with data on a daily (or even weekly) basis.

This is one of the SQL statements we used to create an aggregated view in Google Sheets:

Below is the logic that we applied in R using the **sqldf** library. It enables us to see how many phone screenings, interviews, and offers we had each year in every country:

```
library(sqldf)

aggregated_data = sqldf('SELECT YEAR, COUNTRY, JOB_CATEGORY,
      COUNT(PHONE_SCREENING) AS PHONE_SCREENING,
      COUNT(INTERVIEW) AS INTERVIEW,
      COUNT(OFFER) AS OFFER
      FROM j_h2
      GROUP BY JOB_CATEGORY, YEAR, COUNTRY
      ORDER BY JOB_CATEGORY')
```

This is how the new table “aggregated_data” looks like:

	YEAR	COUNTRY	JOB_CATEGORY	phone_screening	interview	offer
1	2019	Poland	Analysis - other	1	1	0
2	2020	Poland	Analysis - other	1	1	0
3	2020	Poland	Business Analysis	1	1	0
4	2019	Poland	Customer Service	0	0	0
5	2020	Poland	Customer Service	0	0	0
6	2020	Poland	Data Analysis	3	2	0
7	2020	Poland	Data Science	2	0	0
8	2019	Poland	HR	1	0	0
9	2020	Poland	HR	0	0	0
10	2019	Poland	Legal	1	1	0
11	2019	China	Linguistics	0	0	0
12	2019	Ireland	Linguistics	1	1	0
13	2019	Poland	Linguistics	4	2	2
14	2019	Spain	Linguistics	0	0	0
15	2019	United Kingdom	Linguistics	3	2	0
16	2019	United States	Linguistics	1	0	0
17	2020	Poland	Linguistics	2	1	0
18	2020	Slovakia	Linguistics	0	0	0
19	2020	Switzerland	Linguistics	0	0	0
20	2019	Poland	Marketing	0	0	0
21	2020	Poland	Network	1	0	0
22	2019	Poland	Project Management	1	1	0
23	2019	United Kingdom	Project Management	0	0	0
24	2020	Poland	Project Management	1	0	0
25	2020	Poland	QA	3	2	1
26	2020	Poland	SQL	1	1	0
27	2020	Poland	Sales	1	1	1
28	2019	Poland	Teaching	1	0	0
29	2020	Poland	Technical Writing	4	4	2

Adding multiple filters that are conditional can be a very difficult task, but the ShinyWidgets library offers a perfect solution: selectizeGroup-module. Having imported ShinyWidgets, we've replaced ***selectInput()*** with ***selecticizeGroupUI()*** and added one more function – ***callModule()***. This way we have eliminated the possibility of choosing a combination that does not exist. Below you can see the entire solution:

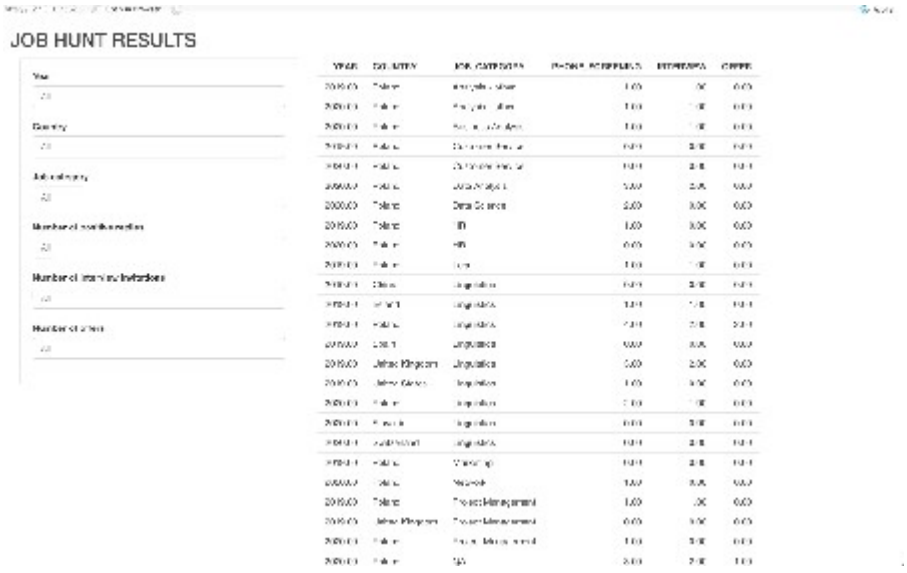
```
library(sqldf)
library(shiny)
library(shinyWidgets)

aggregated_data = sqldf("SELECT YEAR, COUNTRY, JOB_CATEGORY,
      COUNT(PHONE_SCREENING) AS PHONE_SCREENING,
      COUNT(INTERVIEW) AS INTERVIEW,
      COUNT(OFFER) AS OFFER
      FROM j_h2
      GROUP BY JOB_CATEGORY, YEAR, COUNTRY
      ORDER BY JOB_CATEGORY")

shinyApp(
  ui = fluidPage(
    titlePanel("JOB HUNT RESULTS"),
    sidebarPanel(
      selectizeGroupUI(
        id = "fancy_filters",
        inline = FALSE,
```

```
        params = list(
          YEAR = list(inputId = "YEAR", title = "Year", placeholder =
'All'),
          COUNTRY = list(inputId = "COUNTRY", title = "Country",
placeholder = 'All'),
          JOB_CATEGORY = list(inputId = "JOB_CATEGORY", title = "Job
category", placeholder = 'All'),
          PHONE_SCREENING = list(inputId = "PHONE_SCREENING", title =
"Number of positive replies", placeholder = 'All'),
          INTERVIEW = list(inputId = "INTERVIEW", title = "Number of
interview invitations", placeholder = 'All'),
          OFFER = list(inputId = "OFFER", title = "Number of offers",
placeholder = 'All')
        )
      )
    ),
    mainPanel(
      tableOutput("jobhuntData")
    )
  ),

  server = function(input, output, session) {
    res_mod <- callModule(
      module = selectizeGroupServer,
      id = "fancy_filters",
      data = aggregated_data,
      vars = c("YEAR", "COUNTRY", "JOB_CATEGORY", "PHONE_SCREENING",
"INTERVIEW", "OFFER")
    )
    output$jobhuntData <- renderTable({
      res_mod()
    })
  })
```



Conclusion

Working with a new tool like R Shiny can be intimidating at first, but in some ways it can be even easier to learn and understand than Excel or Google Sheets. It is more flexible in terms of adding new features or modifying existing ones. Because we replaced four tables with one, the dashboard not only looks better than our Excel and Google Sheets tool – it is also much easier to use.

Moreover, we managed to create an app where the user is in complete control of the displayed data but does not have access to the backend. This means we do not need to worry about non-technical users making accidental changes to the source code or breaking the app. We can also apply version control and store the source code of the app on services like GitHub in a way that allows us to safely revert to previous versions. This way, anyone who I want to share my code with can download it and make contributions in a controlled environment.