Basic Survey Design in Shiny

Surveys require a range of questions and question types to give the best and most efficient answers. We could just ask our questions in plain text and then provide a text box for users to input as much or as little text details as they want. This however is a very bad approach (unless we want raw text data!). Instead we want to use what we call 'Widgets', they allow specific types of questions to be asked and answered in the best possible way.

This blog post will focus on constructing appropriate tools to implement in our survey environment. This post also serves to demonstrate the versatility of the Shiny package. Links are provided to external site Posit that provide a huge range of possible Widgets. This post will focus on a very brief survey construction UI that shows off the best tools to construct your surveys with. This also demonstrates how much more versatile Shiny is than pre-built survey software like google forms, qualtrics etc.

Here is our full code so you can play around with it as I demonstrate key aspects:

```
library(shiny)

# Shiny app with fields for user input

shinyApp(

ui = fluidPage(

textInput("name", "Name", ""),

checkboxInput("used_shiny", "I've built a Shiny app in R before", FALSE),

sliderInput("r_num_years", "Number of years using R", 0, 25, 2, ticks = FALSE),

dateInput("date", "Date of Birth", value = "2014-01-01"),

checkboxGroupInput("checkGroup", "How did you get to work today?",

choices = list("Walk" = "walk", "Cycle" = "cycle",

"Car" = "car", "Bus" = "bus",

"Tram" = "tram", "Other" = "other")),

dateRangeInput("dates", "How long have you lived in current flat/house?"),

numericInput("num", "Number of Rooms in Flat/House", value = 1),

radioButtons("radio", "Are you enjoying yourself?",

choices = list("YES!" = 1, "Not Sure" = 2, "No :(" = 3),

selected = 0),

checkboxInput("read", "Tick this box if you are completing the survey properly", value = FALSE),

selectInput("select", "What is your favourite area of Scotland?",

choices = list("Highlands and Moray" = 1, "Perthshire" = 2, "North East" = 3,

"Fife and Kinross" = 4, "Edinburgh" = 5, "Borders" = 6,
```

"South" = 7, "Galloway" = 8, "Ayrshire" = 9, "Lanarkshire" = 10,

"Glasgow" = 11, "Forth Valley" = 12, "Inverclyde" = 13),

selected = 1),

fileInput("file", "Upload a funny meme here"),

Submit button

actionButton("submit", "Submit Survey"), # Run JavaScript function on submit

),

server = function(input, output, session) {

})

Each widget will be briefly discussed for your benefit. Note that almost all widgets follow some key constants. The first is that every widget is called using its given name, for example textInput() tells Shiny that the widget we want is a simple text box. Whatever we put inside those () provides detail to expand on that base widget. The second constant is that all widgets require a 'name' so that we can call on them if and when we need (normally in the server). Keeping with the textInput() example, following the open parenthesis we see that this particular widget is called 'name'. After this given name we can now we use a comma and follow up with a label. The label, called 'Name' will be placed above the widget and inform the user what we want them to answer in the text box. This is the requirement for every single widget we use.

After the name and the label we follow with another comma. This last part changes dependent on the type of widget we use but this last part will set the default answer in the question widget. In this example it is kept at "". In other words within the text box widget, when a user opens the app the textbox will be empty. In the example of something like a checkboxInput() this value can be changed to TRUE or FALSE and this will either tick or untick the checkbox. Another example in the case of a radioButtons() will be to use selected = X, where X is one of the pre-defined selections, we normally want to keep that = 0 so we don't unduly influence a user into selecting an answer.

This post will end by a breakdown of each major widget used in this app:

- textInput - Empty textbox input, used to enter someones name
- checkboxInput - Single checkbox input, used to answer a binary question
- sliderInput - A slider bar, used to select a metric value across a specific range
- dateInput - Single date, used to specify a specific date
- checkboxGroupInput - Multiple checkboxes, used to make multiple selections
- dateRangeInput - Multiple dates, used to specific a range of time
- numericInput - Input that only accepts numbers
- radioButtons - Input that gives a selection of options and only one answer allowed
- selectInput - A dropdown of options where only one can be selected, used for long lists

- fileInput - Used to upload a file from your computer

The final part of this app uses what we call an actionButton. This doesn't do anything just yet because we need server side logic to make it do something. When it is clicked in the future we will use this to submit user inputs to our data storage document.

Creating a Web Application in R

This blog post already assumes you have the correct instance of R and Rstudio downloaded and set up on your personal machines.

The Shiny package in R is one of the most versatile packages out there. It allows the user to create their very own web application. That application can take many forms. Two instances of my own Shiny applications are an academic appendix application for conference presentations and a pedagogical teaching tool for students to learn the how social class is operationalised. There are of course a great many other examples (often a lot prettier than my own!). My personal favourites include the Pokemon game and the Covid tracker. Check them out when you have a spare few mins.

This blog post will teach you the very basics of Shiny to get you on your feet. From here you can follow other blog posts set out to enhance your depth of the Shiny environment.

The first thing we need to do is actually install Shiny on our local devices. Remember that Shiny is a package in R. That means we need to install it and 'call' for it from our library. First we:

```
install.packages("shiny")
```

Then in any .R file we have we will load this package at the top by using:

```
library(shiny)
```

Structure:

All Shiny apps are kept within a single app.R file (though we can call functions from other .R files if we want a streamlined app!). This app.R file will need to live in a directory. For this reason we want to create a folder within our local machine and within it deposit the app.R file.

The Shiny app has three primary parts:

- The User interface section
- The Server section
- The call to run the app

The UI or User Interface is the 'face' of the app. It controls what your app looks like. The Server side contains all the 'logic' and background goings on within the app itself. The Server side contains all the 'cool' aspects of what makes an app, an app. Finally, the call to run via the code: shinyApp(ui = ui, server = server)

pulls the components together and explicitly tells Shiny what is what.

The full code of your app.R should look like this:

```
library(shiny)

ui <- ...
```

server <- ...

shinyApp(ui = ui, server = server)

We should load our Shiny package with library(), then our ui section followed by our server section and then finally calling them all together with the shinyApp() function. Remember to save your app each time any changes are made. Rstudio is smart and will know that you are attempting to create an app, in the top right corner there should be a 'runApp' button. This will run a local instance of your app so you can test it prior to placing it online. Sometimes when we add lots of packages to our script this button breaks and we have to do the traditional highlight all then click run (this does the exact same thing).

Let's use an example from the Posit website to see exactly what an App can look like in terms of code. For this example we need another package called 'bslib'. Install before using this code!

library(shiny)

library(bslib)

Define UI for app that draws a histogram ----

ui <- page_sidebar(

App title ----

title = "Hello Shiny!",

Sidebar panel for inputs ----

sidebar = sidebar(

Input: Slider for the number of bins ----

sliderInput(inputId = "bins", label = "Number of bins:", min = 1, max = 50, value = 30)),

Output: Histogram ----

plotOutput(outputId = "distPlot"))

Define server logic required to draw a histogram ----

server <- function(input, output) {

Histogram of the Old Faithful Geyser Data ----

with requested number of bins

This expression that generates a histogram is wrapped in a call # to renderPlot to indicate that:

1. It is "reactive" and therefore should be automatically # re-executed when inputs (input$bins) change

2. Its output type is a plot

output$distPlot <- renderPlot({

x <- faithful$waiting

bins <- seq(min(x), max(x), length.out = input$bins + 1)

```
hist(x, breaks = bins, col = "#007bc2", border = "white", xlab = "Waiting time to next eruption (in mins)", main = "Histogram of waiting times")

})

}

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

Have a play around in this app environment to see what is going on. Look back at your code and see if you can connect what code is doing what thing in the app itself. If this app is a little too complicated, or you prefer seeing a different app use this code instead:

```
# Load the Shiny library

library(shiny)

# Define UI

ui <- fluidPage(

# App title

titlePanel("Basic Shiny App Example"),

# Sidebar layout with input and output

sidebarLayout(

# Sidebar for slider input

sidebarPanel(

sliderInput("num",

"Choose a number:",

min = 1,

max = 100,

value = 50)

),

# Main panel to display the result

mainPanel(

textOutput("result")

)

)

)

# Define server logic

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

```
# Multiply the chosen number by 2 and display it

output$result <- renderText({

paste("Your number multiplied by 2 is:", input$num * 2)

})

}

# Run the application

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

This app uses a slider and whatever number the slider lands on is multiplied by 2. As we can see by these two very simple apps, we can already do very different things with minimal coding.

You have just created your first app! Well done.

Persistent Data Storage Solutions for R Shiny Applications

Shiny apps are very flexible tools. Sometimes however, we want our Shiny apps to store data in a Structured format for survey/questionnaire purposes. Common solutions to writing data in a structured way will not work in a Shiny app because they will only store data locally - think write.csv() etc.

We already have hosting for our apps in Shinyapps.io, but that means our app is now no longer stored locally, it is instead stored on a remote server. If you are a single researcher and are meeting participants face to face (think on street surveys) a local hosting route may be sufficient. If however you are part of a team, or want more flexibility in data storage, remote is the way to go.

This post will cover the most straightforward and popular method of writing remote structured data from a Shiny app to a remote server. This method will use free software and free* hosting services. The software used to store our structured data is Google Sheets (make sure you have an account and are signed up to use it). The R package that we shall be using to store our data to Sheets is called [googlesheets4] (again make sure it is installed). To connect the R code to the app to the Sheets we need to use another Google service called Google Cloud Console.

This solution is popular because we don’t need to deal with the issues of dealing with formal database systems like SQL. Google forms can easily be converted to .csv non-proprietary software for ease of sharing and uploading to our personal favourite analysis software.

First we will start with a super basic Shiny app. Remember to save your app in an appropriate directory where your master folder is named “your_app”, and save your app.R script in here! Our test app will look like this:

```
library(shiny)

# Shiny app with fields for user input

shinyApp(

ui = fluidPage(

DT::dataTableOutput("responses", width = 300), tags$hr(),
```

```
textInput("name", "Name", ""),

checkboxInput("used_shiny", "I've built a Shiny app in R before", FALSE),

sliderInput("r_num_years", "Number of years using R", 0, 25, 2, ticks = FALSE),

actionButton("submit", "Submit")

),

server = function(input, output, session) {

}

)
```

At the moment this is only the shell or UI, the server logic which is where the magic happens will come later.

Setup:

First things first, we need to set up our Google Sheet. Simply create a new sheet, name it whatever suits your project and then make sure to label the headings of your given user input.

Now we have done that we want to return to our simple Shiny app. We need to add a couple of functions to our app in order for it to understand we want all user input to be sent to (write) our Google Sheet when the user hits the Submit button. This is going to require a few functions placed in the server side of the Shiny app (in reality we can remove most of these functions in a real world setting - we are adding a couple more just so you can see in real time what is happening).

```
# Function to save data to Google Sheets

save_data_gsheets <- function(data) {

data <- data %>% as.list() %>% data.frame() # Convert input to a data frame

sheet_append(SHEET_ID, data) # Append data to Google Sheet

}

# Load existing data from Google Sheets

load_data_gsheets <- function() {

read_sheet(SHEET_ID)

}

# Aggregate form data

formData <- reactive({

data <- sapply(fields, function(x) input[[x]])

data

})
```

```
# When submit button is clicked, save form data

observeEvent(input$submit, {

save_data_gsheets(formData())

})

# Show the previous responses in the DataTable

output$responses <- DT::renderDataTable({

input$submit

load_data_gsheets()

})
```

Each function will now be dissected and discussed. Firstly, our save_data_gsheets function is created to save the data input by users to our Google Sheet document. It is telling R to convert the inputs into a data frame (structured data) and then appending that dataframe to our Google Sheets. We tell it to append because we want to add a row to our existing dataframe we don't want to re-write other rows. You may notice that sheet_append is appending something called SHEET_ID. This is something we define outside the Shiny app that is the url for our Google Sheets. We do NOT want others to know this id, this is where all our user data is stored.

Our formData function works in tandem with the prior function by providing our data frame with form or structure. It applies the 'fields' that we define outside the shiny app (this tells us what our variable names are):

```
fields <- c("name", "used_shiny", "r_num_years")
```

Combined these two functions are used in this:

```
# When submit button is clicked, save form data

observeEvent(input$submit, {

save_data_gsheets(formData())

})
```

This piece of code is telling our app that when the submit button is pressed by a user, the save_data_gsheets function is run using the logic formed by our formData function. At its core this is all we need to store data in a Shiny app.

However for this first run through we want to make sure what we are doing is right. We want to see if user data is being stored. For that we use the function:

```
# Load existing data from Google Sheets

load_data_gsheets <- function() {

read_sheet(SHEET_ID)

}
```

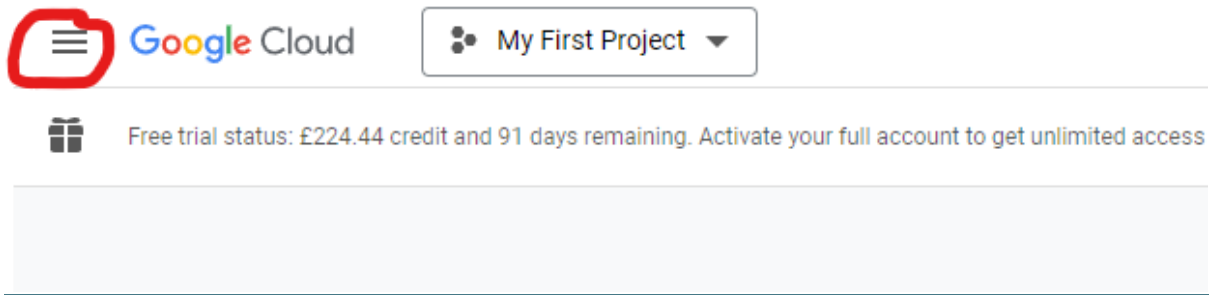
This function loads the Google Sheet in the Shiny app. Combined with this:

```
# Show the previous responses in the DataTable

output$responses <- DT::renderDataTable({

input$submit

load_data_gsheets()

})
```

Set of R code, the Google Sheet is rendered appropriately within the Shiny app environment.

This is unfortunately not the last step. If we were to run this app we would quickly run into errors. Trying to edit a Google Form from someone's account (even if it is your own) is not something we can just do. We require some form of authentication. This would be really easy if only we wanted to edit the sheet but we want all users to 'edit'.

To authenticate all user data being edited in our constructed Google Sheet we need to create an account on Google Cloud Console. The account can be created easily enough, though you will need to provide credit card authentication (this is a free service - you will not be charged!). After you have done this, go to the navigation menu, go to 'IAM and Admin' then Service Account.



You need to now create a Service Account and assign it a name (this is so Google Sheets has perms). You can leave the Service Account access blank. Under keys for your new account, you need to create a new key. Make sure this key is in JSON format! Download this JSON file.

Open the JSON file - if you don't have a dedicated reader, Rstudio works fine. You want to copy the client email from this document and then head back over to your Google Sheet tab. From here we want to do two things, first go to Share and change general access to anyone with the link to editor. Secondly, add people by pasting that email you copied and send. After this head back over to your Google Cloud tab and enable Sheet perms.

Now we want to head over to the directory where our Shiny app is stored. We want to create a folder called ".secrets". We name it that because no one except you should ever see the details inside! Open that folder and then paste your JSON file into it. Rename that file something simple like "service-account.json". We do this because we need to paste this in our Shiny file and a long filename just doesn't look nice.

This should be us all done, and our updated Shiny code should look like this:

```
library(shiny)
```

```
library(googlesheets4)
library(DT)
# Authenticate using the service account JSON key
gs4_auth(path = ".secrets/service-account.json")
# Define the fields to save from the form
fields <- c("name", "used_shiny", "r_num_years")
SHEET_ID <- "https://docs.google.com/spreadsheets/inserthere"
# Shiny app with fields for user input
shinyApp(
ui = fluidPage(
DT::dataTableOutput("responses", width = 300), tags$hr(),
textInput("name", "Name", ""),
checkboxInput("used_shiny", "I've built a Shiny app in R before", FALSE),
sliderInput("r_num_years", "Number of years using R", 0, 25, 2, ticks = FALSE),
actionButton("submit", "Submit")
),
server = function(input, output, session) {
# Function to save data to Google Sheets
save_data_gsheets <- function(data) {
data <- data %>% as.list() %>% data.frame() # Convert input to a data frame
sheet_append(SHEET_ID, data) # Append data to Google Sheet
}
# Load existing data from Google Sheets
load_data_gsheets <- function() {
read_sheet(SHEET_ID)
}
# Aggregate form data
formData <- reactive({
data <- sapply(fields, function(x) input[[x]])
data
})
```

```
# When submit button is clicked, save form data

observeEvent(input$submit, {

save_data_gsheets(formData())

})

# Show the previous responses in the DataTable

output$responses <- DT::renderDataTable({

input$submit

load_data_gsheets()

})

}

)
```

The last thing we need to do is deploy our app. We do that through the publish button when we locally test run our app or by pasting the following code in the console (only use this method if you run into issues with publish button. This sometimes occurs because Shinyapps.io things you are trying to upload a quatro doc).

rsconnect::deployApp(appDir = "directory", appPrimaryDoc = "appname.R", appFiles = c("testapp.R", ".secrets/"))

Advanced Shiny Applications

We could stop right here. We have our app, it has its server logic, and user inputs are sent to our Google docs. But… our apps look a little… naff. This is where advanced Shiny techniques come in. This blog post will help you tweak your app to make it look a lot nicer and more 'app like' and provide advanced server logic to make your survey work better.

The easiest step shall be taken first. As you can see, when you run your app it looks fine, but everything is on the left hand side, and there is not much to really look at. This is when we turn to Javascript to help us out. Don't worry, there isn't anything major to learn or even remember here. All this post will teach you is the basics that can be applied to any App you make. We want to make our app look decent. Firstly we want to centre all our content so it actually looks like a proper form to fill out. Then we want to set a font style and size dependent on the type of object we are working with. We also want to set a uniform background colour. Another important survey aspect is a mandatory question - for that we will need a little red asterisk next to every question we want the user to answer.

For all of this we need to first install the shinyjs package. This allows Shiny and javascript to work together in relative harmony. After we have done this and added it to our growing library of packages we can get to work. Within our ui section of our App we want to include the following code:

useShinyjs(), # Enable shinyjs for JavaScript capabilities

Following this we want to create a ‘style guide’ for all our UI elements. we capture all this within the code:
tags$style(HTML(""")

This tells Shiny that we are create tags (these are things we can apply to specific widgets/objects in our App environment) based in HTML style, and “” the Javascript is contained within. We want to create six different tags:

```
.survey-container {
max-width: 600px;
margin: auto;
padding: 20px;
font-family: Arial, sans-serif;
}
.survey-question {
font-size: 18px;
font-weight: bold;
margin-top: 20px;
}
.survey-input {
margin-top: 10px;
}
.btn-primary {
background-color: #4CAF50;
color: white;
border: none;
padding: 10px 20px;
font-size: 16px;
}
.submit-container {
text-align: center;
margin-top: 30px;
}
.mandatory-asterisk {
color: red;
```

margin-left: 5px;

font-weight: bold;

}

The first of these is our survey-container. This is applied to our overall survey panel. We are telling Shiny that for anything with this tag, the max width of any object is restricted to 600px, has an automatic margin, with some padding of 20px, and everything within uses the sans-serif font Arial.

The next is the survey-question tag. This tells anything associated with the tag to be bold size 18 font with 20px margin at the top. The survey-input tag only applies a 10px margin at the top. The btn-primary gives the submit button its colour, font, and size. The submit-container tag gives the alignment of text within the button and top margin pixels. Finally our mandatory-asterix tag makes any text within it bold and red.

To apply these tags to our App we have to enclose items within a div(class = *insert tag here*). This is what that looks like in our transformed UI:

Survey Questions

div(class = "survey-question", HTML("Name*")),

div(class = "survey-input", textInput("name", label = NULL, placeholder = "Enter your name")),

div(class = "survey-question", HTML("Date of Birth*")),

div(class = "survey-input", dateInput("date", label = NULL, value = "2014-01-01")),

div(class = "survey-question", HTML("How did you travel here today? (Tick all that apply)*")),

div(class = "survey-input", checkboxGroupInput("checkGroup", label = NULL,

choices = list("Walk" = "walk", "Cycle" = "cycle",

"Car" = "car", "Bus" = "bus",

"Tram" = "tram", "Other" = "other"))),

div(class = "survey-question", HTML("Number of years using R*")),

div(class = "survey-input", sliderInput("r_num_years", label = NULL, min = 0, max = 25, value = 2, ticks = FALSE)),

div(class = "survey-question", HTML("Approximately how long have you lived at your current address?*")),

div(class = "survey-input", dateRangeInput("dates", label = NULL)),

div(class = "survey-question", HTML("How many rooms does your flat/house have?*")),

div(class = "survey-input", numericInput("num", label = NULL, value = 1)),

```
div(class = "survey-question", HTML("This course seems to be worth my time<span class='mandatory-asterisk'>*</span>")),

div(class = "survey-input", radioButtons("radio", label = NULL,

choices = list("YES!" = 1, "Not Sure" = 2, "No :(" = 3),

selected = 0)),

div(class = "survey-question", HTML("Tick if you have read this message<span class='mandatory-asterisk'>*</span>")),

div(class = "survey-input", checkboxInput("read", label = NULL, value = FALSE)),

div(class = "survey-question", HTML("Select your favorite region of Scotland<span class='mandatory-asterisk'>*</span>")),

div(class = "survey-input", selectInput("select", label = NULL,

choices = list("Highlands and Moray" = 1, "Perthshire" = 2, "North East" = 3,

"Fife and Kinross" = 4, "Edinburgh" = 5, "Borders" = 6,

"South" = 7, "Galloway" = 8, "Ayrshire" = 9, "Lanarkshire" = 10,

"Glasgow" = 11, "Forth Valley" = 12, "Inverclyde" = 13),

selected = 1)),

div(class = "survey-question", "Please provide a funny meme/cute picture of an animal (optional)"),

div(class = "survey-input", fileInput("file", label = NULL)),

# Submit button

div(class = "submit-container",

actionButton("submit", "Submit Survey", class = "btn-primary")
```

Notice that before each Widget had a self contained name, label, and selection. Now we have two separate div() for each widget. The first is the user text or question text. Notice all of these are using the div(class = “survey-question”, “X“). This is then followed by a second div that uses ‘survey-input’ for the actual widgets. We also include our mandatory asterisk within the first div(). Here is a snippet of our code to explain:

```
div(class = "survey-question", HTML("Name<span class='mandatory-asterisk'>*</span>")),
```

Note that within the survey-question tag we have a second mandatory-asterisk tag. For this to work we have to use some HTML language called ‘span’.

Advanced Shiny Logic

We could stop right here. We have our app, it has its server logic, and user inputs are sent to our Google docs. But… our apps look a little… naff. This is where advanced Shiny techniques come in. This blog post will help you tweak your app to make it look a lot nicer and more ‘app like’ and provide advanced server logic to make your survey work better.

The easiest step shall be taken first. As you can see, when you run your app it looks fine, but everything is on the left hand side, and there is not much to really look at. This is when we turn to Javascript to help us out. Don't worry, there isn't anything major to learn or even remember here. All this post will teach you is the basics that can be applied to any App you make. We want to make our app look decent. Firstly we want to centre all our content so it actually looks like a proper form to fill out. Then we want to set a font style and size dependent on the type of object we are working with. We also want to set a uniform background colour. Another important survey aspect is a mandatory question - for that we will need a little red asterisk next to every question we want the user to answer.

For all of this we need to first install the shinyjs package. This allows Shiny and javascript to work together in relative harmony. After we have done this and added it to our growing library of packages we can get to work. Within our ui section of our App we want to include the following code:

useShinyjs(), # Enable shinyjs for JavaScript capabilities

Following this we want to create a 'style guide' for all our UI elements. we capture all this within the code:
tags$style(HTML(""")

This tells Shiny that we are create tags (these are things we can apply to specific widgets/objects in our App environment) based in HTML style, and "" the Javascript is contained within. We want to create six different tags:
.survey-container {

max-width: 600px;

margin: auto;

padding: 20px;

font-family: Arial, sans-serif;

}

.survey-question {

font-size: 18px;

font-weight: bold;

margin-top: 20px;

}

.survey-input {

margin-top: 10px;

}

.btn-primary {

```
background-color: #4CAF50;

color: white;

border: none;

padding: 10px 20px;

font-size: 16px;

}

.submit-container {

text-align: center;

margin-top: 30px;

}

.mandatory-asterisk {

color: red;

margin-left: 5px;

font-weight: bold;

}
```

The first of these is our survey-container. This is applied to our overall survey panel. We are telling Shiny that for anything with this tag, the max width of any object is restricted to 600px, has an automatic margin, with some padding of 20px, and everything within uses the sans-serif font Arial.

The next is the survey-question tag. This tells anything associated with the tag to be bold size 18 font with 20px margin at the top. The survey-input tag only applies a 10px margin at the top. The btn-primary gives the submit button its colour, font, and size. The submit-container tag gives the alignment of text within the button and top margin pixels. Finally our mandatory-asterix tag makes any text within it bold and red.

To apply these tags to our App we have to enclose items within a div(class = *insert tag here*). This is what that looks like in our transformed UI:

```
# Survey Questions

div(class = "survey-question", HTML("Name<span class='mandatory-asterisk'>*</span>")),

div(class = "survey-input", textInput("name", label = NULL, placeholder = "Enter your name")),

div(class = "survey-question", HTML("Date of Birth<span class='mandatory-
asterisk'>*</span>")),

div(class = "survey-input", dateInput("date", label = NULL, value = "2014-01-01")),

div(class = "survey-question", HTML("How did you travel here today? (Tick all that apply)<span
class='mandatory-asterisk'>*</span>")),
```

```
div(class = "survey-input", checkboxGroupInput("checkGroup", label = NULL,

choices = list("Walk" = "walk", "Cycle" = "cycle",

"Car" = "car", "Bus" = "bus",

"Tram" = "tram", "Other" = "other"))),

div(class = "survey-question", HTML("Number of years using R<span class='mandatory-asterisk'>*</span>")),

div(class = "survey-input", sliderInput("r_num_years", label = NULL, min = 0, max = 25, value = 2, ticks = FALSE)),

div(class = "survey-question", HTML("Approximately how long have you lived at your current address?<span class='mandatory-asterisk'>*</span>")),

div(class = "survey-input", dateRangeInput("dates", label = NULL)),

div(class = "survey-question", HTML("How many rooms does your flat/house have?<span class='mandatory-asterisk'>*</span>")),

div(class = "survey-input", numericInput("num", label = NULL, value = 1)),

div(class = "survey-question", HTML("This course seems to be worth my time<span class='mandatory-asterisk'>*</span>")),

div(class = "survey-input", radioButtons("radio", label = NULL,

choices = list("YES!" = 1, "Not Sure" = 2, "No :(" = 3),

selected = 0)),

div(class = "survey-question", HTML("Tick if you have read this message<span class='mandatory-asterisk'>*</span>")),

div(class = "survey-input", checkboxInput("read", label = NULL, value = FALSE)),

div(class = "survey-question", HTML("Select your favorite region of Scotland<span class='mandatory-asterisk'>*</span>")),

div(class = "survey-input", selectInput("select", label = NULL,

choices = list("Highlands and Moray" = 1, "Perthshire" = 2, "North East" = 3,

"Fife and Kinross" = 4, "Edinburgh" = 5, "Borders" = 6,

"South" = 7, "Galloway" = 8, "Ayrshire" = 9, "Lanarkshire" = 10,

"Glasgow" = 11, "Forth Valley" = 12, "Inverclyde" = 13),

selected = 1)),

div(class = "survey-question", "Please provide a funny meme/cute picture of an animal (optional)"),

div(class = "survey-input", fileInput("file", label = NULL)),

# Submit button
```

div(class = "submit-container",

actionButton("submit", "Submit Survey", class = "btn-primary")

Notice that before each Widget had a self contained name, label, and selection. Now we have two separate div() for each widget. The first is the user text or question text. Notice all of these are using the div(class = “survey-question”, “X“). This is then followed by a second div that uses ‘survey-input’ for the actual widgets. We also include our mandatory asterisk within the first div(). Here is a snippet of our code to explain:
div(class = "survey-question", HTML("Name*")),

Note that within the survey-question tag we have a second mandatory-asterisk tag. For this to work we have to use some HTML language called ‘span’.

Advanced Shiny Logic

Our App now actually looks App-like which is nice. There are a few extra things we can (and in some cases should do). The first is relevant to Ethics. As we all should know an individual should be provided with an ethics related statement prior to conducting any survey - this statement looks somewhat different from institution to institution but follows similar beats. What we need is for users to be able to fully consent to their responses being collected. For that we need a page before these even see the survey where the ins and outs are explained, they user can give or retract consent, and the app can respond to those needs.

For our app we are going to create a ‘consent’ page to start off. This will have some explainer text and a consent box for users to click or not click. There will be a button to press for users to move on. If consent is not given users will be taken to a thank you page where they can then close the tab. If consent is given they will then be taken to the survey page. We will also update our Javascript logic to make everything look consistent.

We also want to implement some rather advanced features to our Survey to help us out in the long run. Remember we added those nice little red asterisks to our questions? Well that is all well and good but we want to actually make users answer them. For this, we are going to implement some server logic to stop the submit button being pressed until all required questions are answered - Indicated via the red asterisk.

Finally, and this is just to show off, but a common issue with surveys taken online is that it is hard to assess if a user has actually filled it out properly. We have already tried to combat this with the simple checkbox question to ‘tick if read properly’, we also have now implemented the ‘submit only when completed all questions’ logic. Now we are going to implement some behind the scenes ‘magic’. We are going to use some javascript code to record in seconds how long it takes a user from page start to button submit to finish our survey. This will then be deposited in our user input sheet. This can tell us if a user (if they even are a user and not a bot) has completed our survey in an appropriate amount of time.

All of this logic can be found in our final form code:

library(shiny)

library(shinyjs)

library(googlesheets4)

```
library(DT)

# Authenticate using the service account JSON key

gs4_auth(path = "/Users/scottoatley/Documents/testapp/secrets/service-account.json")

# Define the fields to save from the form (adding columns for each checkbox option)

fields <- c("name", "dob", "r_num_years", "rooms", "time_worth", "read",

"region", "file", "walk", "cycle", "car", "bus", "tram", "other")

SHEET_ID <- "https://docs.google.com/spreadsheets/d/1rhSUw2H-SQb66Wd3eBjHeRa-ZUnyS8N91YR9y13ln38/edit?gid=0#gid=0"

# Shiny app with fields for user input

shinyApp(

ui = fluidPage(

useShinyjs(), # Enable shinyjs for JavaScript capabilities

tags$style(HTML("

.survey-container {

max-width: 600px;

margin: auto;

padding: 20px;

font-family: Arial, sans-serif;

}

.survey-question {

font-size: 18px;

font-weight: bold;

margin-top: 20px;

}

.survey-input {

margin-top: 10px;

}

.btn-primary {

background-color: #4CAF50;

color: white;

border: none;
```

```
padding: 10px 20px;
font-size: 16px;
}
.submit-container {
text-align: center;
margin-top: 30px;
}
.mandatory-asterisk {
color: red;
margin-left: 5px;
font-weight: bold;
}
.thank-you {
text-align: center;
font-size: 24px;
font-weight: bold;
margin-top: 50px;
}
")),
tags$script(HTML("
// JavaScript to start timer when the user opens the app
var startTime;
$(document).on('shiny:connected', function() {
startTime = new Date().getTime(); // Capture the current time when the app is opened
});
// JavaScript function to calculate elapsed time on submit
function getElapsedTime() {
var endTime = new Date().getTime();
var timeSpent = (endTime - startTime) / 1000; // Time in seconds
Shiny.setInputValue('time_spent', timeSpent); // Send to Shiny input
}
```

")),

div(class = "survey-container",

Consent Screen UI

conditionalPanel(

condition = "output.showConsentScreen == true",

h2("Survey Consent Form", align = "center"),

p("Thank you for considering participating in our survey. This survey is designed to understand your preferences and experiences."),

p("Your responses will be anonymized and used solely for academic purposes. Please indicate if you are comfortable proceeding."),

checkboxInput("consent", "I agree to participate in the survey and understand my data will be anonymized.", value = FALSE),

actionButton("consent_submit", "Continue", class = "btn-primary")

),

Survey Form UI (shown after consent)

conditionalPanel(

condition = "output.showSurveyForm == true",

h2("Survey Form", align = "center"),

Survey Questions

div(class = "survey-question", HTML("Name*")),

div(class = "survey-input", textInput("name", label = NULL, placeholder = "Enter your name")),

div(class = "survey-question", HTML("Date of Birth*")),

div(class = "survey-input", dateInput("date", label = NULL, value = "2014-01-01")),

div(class = "survey-question", HTML("How did you travel here today? (Tick all that apply)*")),

div(class = "survey-input", checkboxGroupInput("checkGroup", label = NULL,

choices = list("Walk" = "walk", "Cycle" = "cycle",

"Car" = "car", "Bus" = "bus",

"Tram" = "tram", "Other" = "other"))),

div(class = "survey-question", HTML("Number of years using R*")),

```
div(class = "survey-input", sliderInput("r_num_years", label = NULL, min = 0, max = 25, value = 2,
ticks = FALSE)),

div(class = "survey-question", HTML("Approximately how long have you lived at your current
address?<span class='mandatory-asterisk'>*</span>")),

div(class = "survey-input", dateRangeInput("dates", label = NULL)),

div(class = "survey-question", HTML("How many rooms does your flat/house have?<span
class='mandatory-asterisk'>*</span>")),

div(class = "survey-input", numericInput("num", label = NULL, value = 1)),

div(class = "survey-question", HTML("This course seems to be worth my time<span
class='mandatory-asterisk'>*</span>")),

div(class = "survey-input", radioButtons("radio", label = NULL,

choices = list("YES!" = 1, "Not Sure" = 2, "No :(" = 3),

selected = 0)),

div(class = "survey-question", HTML("Tick if you have read this message<span
class='mandatory-asterisk'>*</span>")),

div(class = "survey-input", checkboxInput("read", label = NULL, value = FALSE)),

div(class = "survey-question", HTML("Select your favorite region of Scotland<span
class='mandatory-asterisk'>*</span>")),

div(class = "survey-input", selectInput("select", label = NULL,

choices = list("Highlands and Moray" = 1, "Perthshire" = 2, "North East" = 3,

"Fife and Kinross" = 4, "Edinburgh" = 5, "Borders" = 6,

"South" = 7, "Galloway" = 8, "Ayrshire" = 9, "Lanarkshire" = 10,

"Glasgow" = 11, "Forth Valley" = 12, "Inverclyde" = 13),

selected = 1)),

div(class = "survey-question", "Please provide a funny meme/cute picture of an animal
(optional)"),

div(class = "survey-input", fileInput("file", label = NULL)),

# Submit button

div(class = "submit-container",

actionButton("submit", "Submit Survey", class = "btn-primary", onclick = "getElapsedTime()") #
Run JavaScript function on submit

)

),

# Thank-You Message UI (shown if consent is not given)
```

```
conditionalPanel(
condition = "output.showThankYou == true",
div(class = "thank-you", "Thank you for your time. Have a great day!")
),
# Section to show responses after submission
DT::dataTableOutput("responses", width = "50%")
)
),
server = function(input, output, session) {
# Flags to manage screen visibility
showConsentScreen <- reactiveVal(TRUE)
showSurveyForm <- reactiveVal(FALSE)
showThankYou <- reactiveVal(FALSE)
# Update output to reflect flags
output$showConsentScreen <- reactive(showConsentScreen())
output$showSurveyForm <- reactive(showSurveyForm())
output$showThankYou <- reactive(showThankYou())
outputOptions(output, "showConsentScreen", suspendWhenHidden = FALSE)
outputOptions(output, "showSurveyForm", suspendWhenHidden = FALSE)
outputOptions(output, "showThankYou", suspendWhenHidden = FALSE)
# Consent button handling
observeEvent(input$consent_submit, {
if (input$consent) {
showConsentScreen(FALSE)
showSurveyForm(TRUE)
} else {
showConsentScreen(FALSE)
showThankYou(TRUE)
}
})
# Disable the submit button initially
```

```
shinyjs::disable("submit")
# Reactive expression to check if all required fields are filled
allFilled <- reactive({
!is.null(input$name) && input$name != "" &&
!is.null(input$date) &&
!is.null(input$checkGroup) && length(input$checkGroup) > 0 &&
!is.null(input$r_num_years) &&
!is.null(input$num) && input$num > 0 &&
!is.null(input$radio) &&
!is.null(input$select)
})
# Observe if all required fields are filled and enable the submit button
observe({
if (allFilled()) {
shinyjs::enable("submit")
} else {
shinyjs::disable("submit")
}
})
# Function to save data to Google Sheets
save_data_gsheets <- function(data) {
data <- data %>% as.list() %>% data.frame() # Convert input to a data frame
sheet_append(SHEET_ID, data) # Append data to Google Sheet
}
# Load existing data from Google Sheets
load_data_gsheets <- function() {
read_sheet(SHEET_ID)
}
# Aggregate form data and handle multi-select fields
formData <- reactive({
# Set each travel option as 1 (selected) or 0 (not selected)
```

```
travel_choices <- c("walk", "cycle", "car", "bus", "tram", "other")

travel_data <- setNames(as.list(ifelse(travel_choices %in% input$checkGroup, 1, 0)),
travel_choices)

# Separate start and end dates

date_start <- if (!is.null(input$dates)) input$dates[1] else NA

date_end <- if (!is.null(input$dates)) input$dates[2] else NA

# Create the final data frame

data <- list(

name = input$name,

dob = input$date,

r_num_years = input$r_num_years,

rooms = input$num,

time_worth = input$radio,

read = as.numeric(input$read), # Convert logical to numeric (1 = TRUE, 0 = FALSE)

region = input$select,

file = if (!is.null(input$file)) input$file$name else NA, # Handle file name

time_spent = input$time_spent, # Include the time spent on the form

address_start = date_start,

address_end = date_end

)

# Combine data with travel_data

data <- c(data, travel_data)

# Convert list to data frame with one row

data.frame(data)

})

# When submit button is clicked, save form data and reload responses

observeEvent(input$submit, {

save_data_gsheets(formData())

# Refresh responses

output$responses <- DT::renderDataTable({

load_data_gsheets()
```

```
})
})
}
)
```