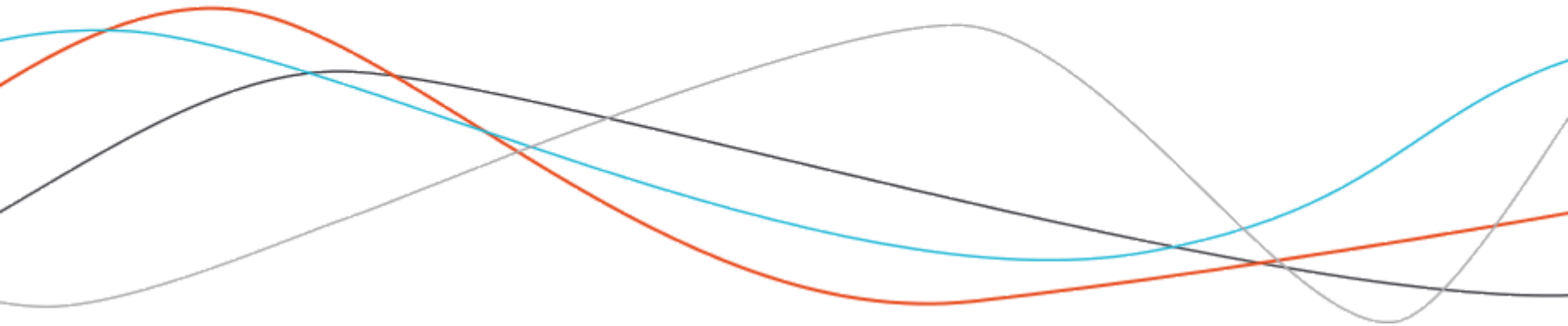
THINKR

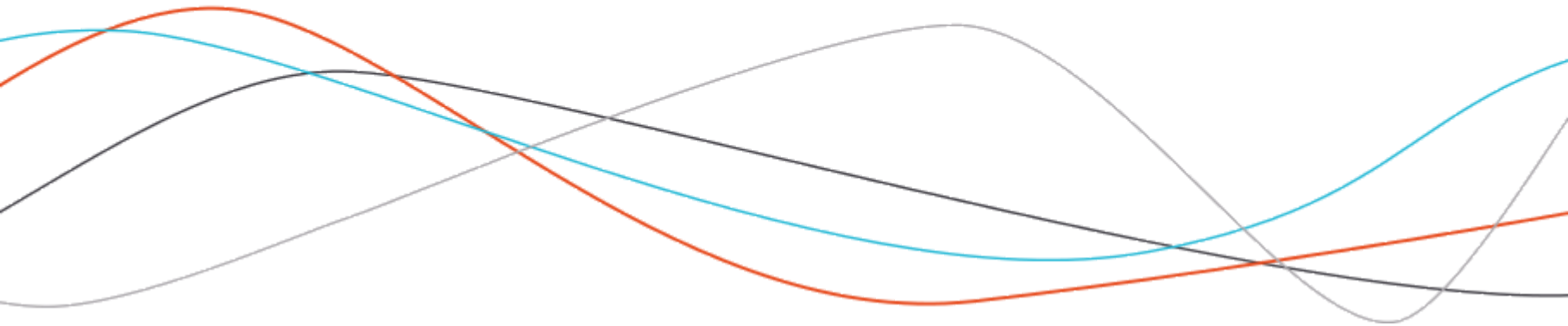
Building Successful Shiny Apps with {golem}

Colin Fay - ThinkR

PART 02 - DEV



STRUCTURING YOUR APPLICATION



Application logic vs business logic

- Application logic: the things that make your Shiny app interactive
- Business logic: core algorithms and functions that make your application specific to your area of work

Keep them separated!

Structuring your app

Naming convention:

- `app_*`: global infrastructure functions
- `fct_*`: business logic functions
- `mod_*`: file with ONE module (ui + server)
- `utils_*`: cross module utilitarian functions
- `*_ui_*` / `*_server_*`: relates to UI and SERVER

Why bother?

- 🤔: `R/connect.R`
- 🤔: `R/summary.R`
- 🤔: `R/plot.R`
- 🤔: `R/odbc.R`

Why bother?

- 🤔: `R/connect.R`
- 🤔: `R/summary.R`
- 🤔: `R/plot.R`
- 🤔: `R/odbc.R`

- 🤩: `R/fct_connect.R`
- 🤩: `R/mod_summary.R`
- 🤩: `R/utils_plot.R`
- 🤩: `R/fct_odbc.R`

Why bother?

Separation of business & app logic is crucial:

- Easier to work on functions when they are not inside the app (you don't need to relaunch the app every time)
- You can test your business logic with classical package testing tools
- You can use the business logic functions outside the app

Structuring your app

With your nearest neighbor, think about where to put these functions (and how to name them):

- A module that contains the "About" panel of the App
- A function that takes a number `n`, a table, and return the `n` first rows of the table
- A function that takes a string, and returns this string as an html tag colored in red.
- A function that connects to the mongo database

05:00

02_dev.R

```
golem::add_fct( "helpers" )
golem::add_utils( "data_ui" )
```

- Go to R/fct_helpers.R
- Go to R/utils_data_ui.R

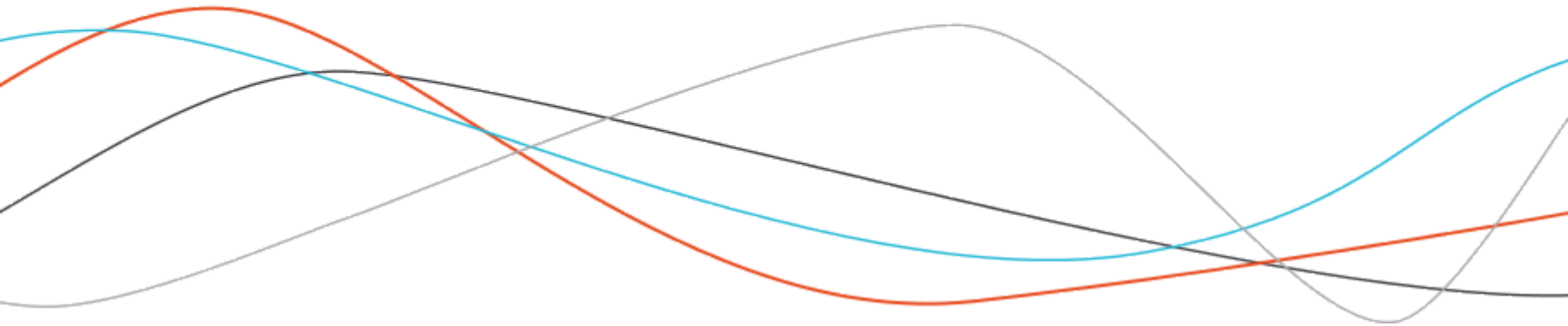
Creates `fct_*` and `utils_*` files

```
golem::add_module( name = "my_first_module")
```

✓ File created at R/mod_my_first_module.R
- Go to R/mod_my_first_module.R

Builds a skeleton for a Shiny Module

USING SHINY MODULES



About Shiny Modules

- "Pieces" of Shiny Apps
- Always come in pair (UI + Server)
- Manage the unique IDs necessity of Shiny
- "Functionnalize" your app

One million "Validate" buttons

```
library(shiny)
ui <- function(request){
  fluidPage(
    sliderInput("choice1", "choice 1", 1, 10, 5),
    actionButton("validate1", "Validate choice 1"),
    sliderInput("choice2", "choice 2", 1, 10, 5),
    actionButton("validate2", "Validate choice 2")
  )
}
server <- function(input, output, session){
  observeEvent( input$validate1 , {
    print(input$choice1)
  })
  observeEvent( input$validate2 , {
    print(input$choice2)
  })
}
shinyApp(ui, server)
```

Why Shiny Modules

Functionalizing Shiny elements:

```
name_ui <- function(id){
  ns <- NS(id)
  tagList(
    sliderInput(ns("choice"), "Choice", 1, 10, 5),
    actionButton(ns("validate"), "Validate Choice")
  )
}

name_server <- function(input, output, session){

  observeEvent( input$validate , {
    print(input$choice)
  })

}
```

Why Shiny Modules

Functionalizing Shiny elements:

```
library(shiny)
ui <- function(request){
  fluidPage(
    name_ui("name_ui_1"),
    name_ui("name_ui_2")
  )
}

server <- function(input, output, session){
  callModule(name_server, "name_ui_1")
  callModule(name_server, "name_ui_2")
}

shinyApp(ui, server)
```

Why Shiny Modules

```
id <- "name_ui_1"
ns <- NS(id)
ns("choice")
```

```
[1] "name_ui_1-choice"
```

```
name_ui <- function(id, butname){
  ns <- NS(id)
  tagList( actionButton(ns("validate"), butname) )
}

name_ui("name_ui_1", "Validate Choice")
name_ui("name_ui_2", "Validate Choice, again")
```

```
<button id="name_ui_1-validate" type="button" class="btn btn-default action-
button">Validate Choice</button>

<button id="name_ui_2-validate" type="button" class="btn btn-default action-
button">Validate Choice, again</button>
```

02_dev.R

```
golem::add_module( name = "my_first_module")
```

Builds a skeleton for a Shiny Module

02_dev.R

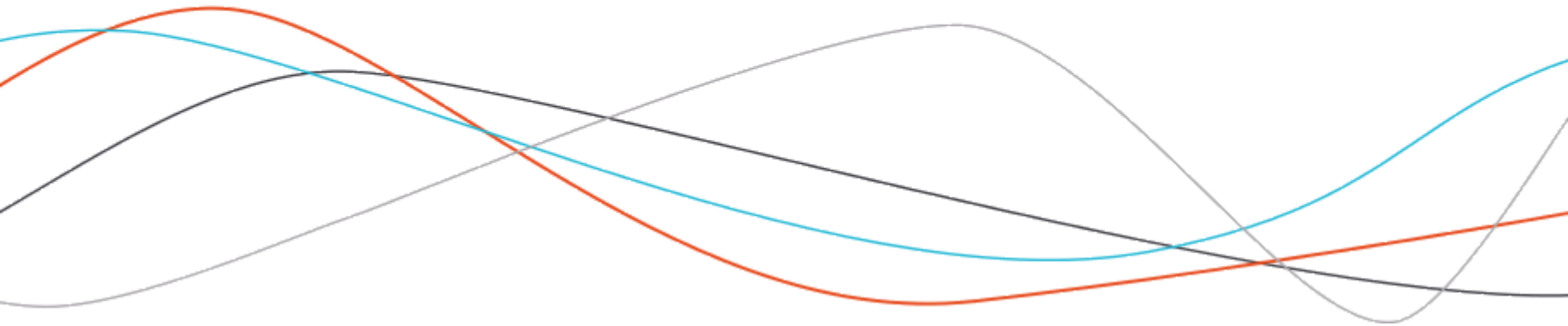
```
#' my_first_module UI Function
#'
#' @description A shiny Module.
#'
#' @param id,input,output,session Internal parameters for {shiny}.
#'
#' @noRd
#'
#' @importFrom shiny NS tagList
mod_my_first_module_ui <- function(id){
  ns <- NS(id)
  tagList(

  )
}
```

02_dev.R

```
#' my_first_module Server Function
#'
#' @noRd
mod_my_first_module_server <- function(input, output, session){
  ns <- session$ns

}

## To be copied in the UI
# mod_my_first_module_ui("my_first_module_ui_1")

## To be copied in the server
# callModule(mod_my_first_module_server, "my_first_module_ui_1")
```

⚠️ DON'T FORGET THE NS() IN THE UI ⚠️

UNDERSTANDING NAMESPACE & DEPENDENCIES



About the NAMESPACE file

- One of the most important files of your package
- ⚠️ NEVER EDIT BY HAND ⚠️
- Describes **how your package interacts with R, and with other packages**
- Lists functions that are exported (from your app) and imported (from other packages)

What's a dependency?

- Your app needs external functions (at least from `{shiny}`)
- The `DESCRIPTION` file contains the package dependencies
- Are added to the `DESCRIPTION` with:

```
usethis::use_package("attempt")
```

What's a dependency?

- You also need to add tags on top of each functions that specify what deps are imported
- Either with `@import` (a whole package) and `@importFrom` (a specific function).

> golem built modules, by default, import elements from `{shiny}`:

```
#' @importFrom shiny NS tagList
```

Do this for EACH function

```
#' @import magrittr
#' @importFrom  stats na.omit

mean_no_na <- function(x){
  x <- x %>% na.omit()
  sum(x)/length(x)
}
```

- You can use `import` or `importFrom`.
- The better is to use `importFrom`, for preventing namespace conflict.
- Add to EACH function.

> It will take some time, but it's better on the long run.

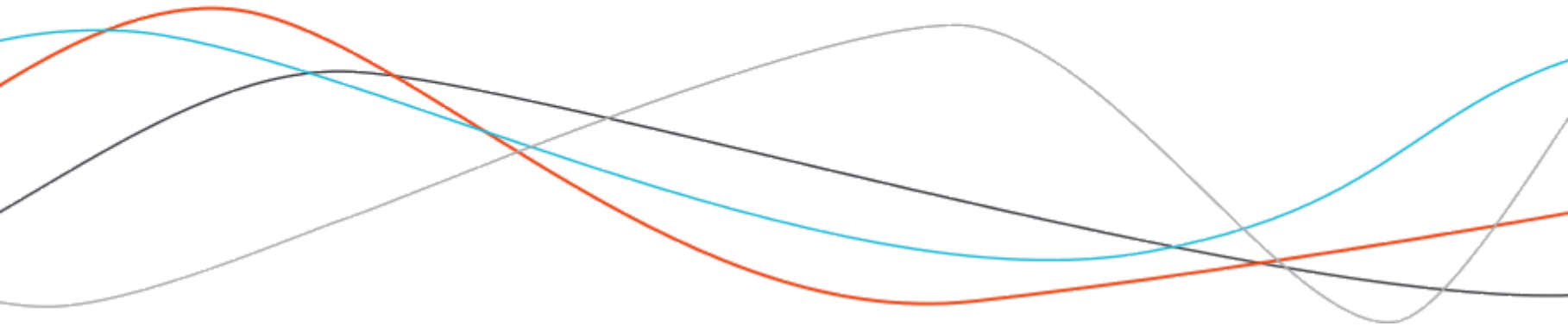
How many dependencies?

There is no perfect answer.

Just keep in mind that depending on another package means that :

- you will potentially have to recode your package if some breaking change happen in your dependencies
- If one of your dependencies is removed from CRAN, you will be removed too
- The more you have deps, the longer it takes to deploy the app

ADDING INTERNAL DATASETS



Adding datasets

- You might need internal data inside your app
- Call `usethis::use_data_raw` from `dev/02_dev.R`

```
## Add internal datasets ----
## If you have data in your package
usethis::use_data_raw( name = "dataset" )
```

- Creates `data-raw/dataset.R`
- Inside this file:

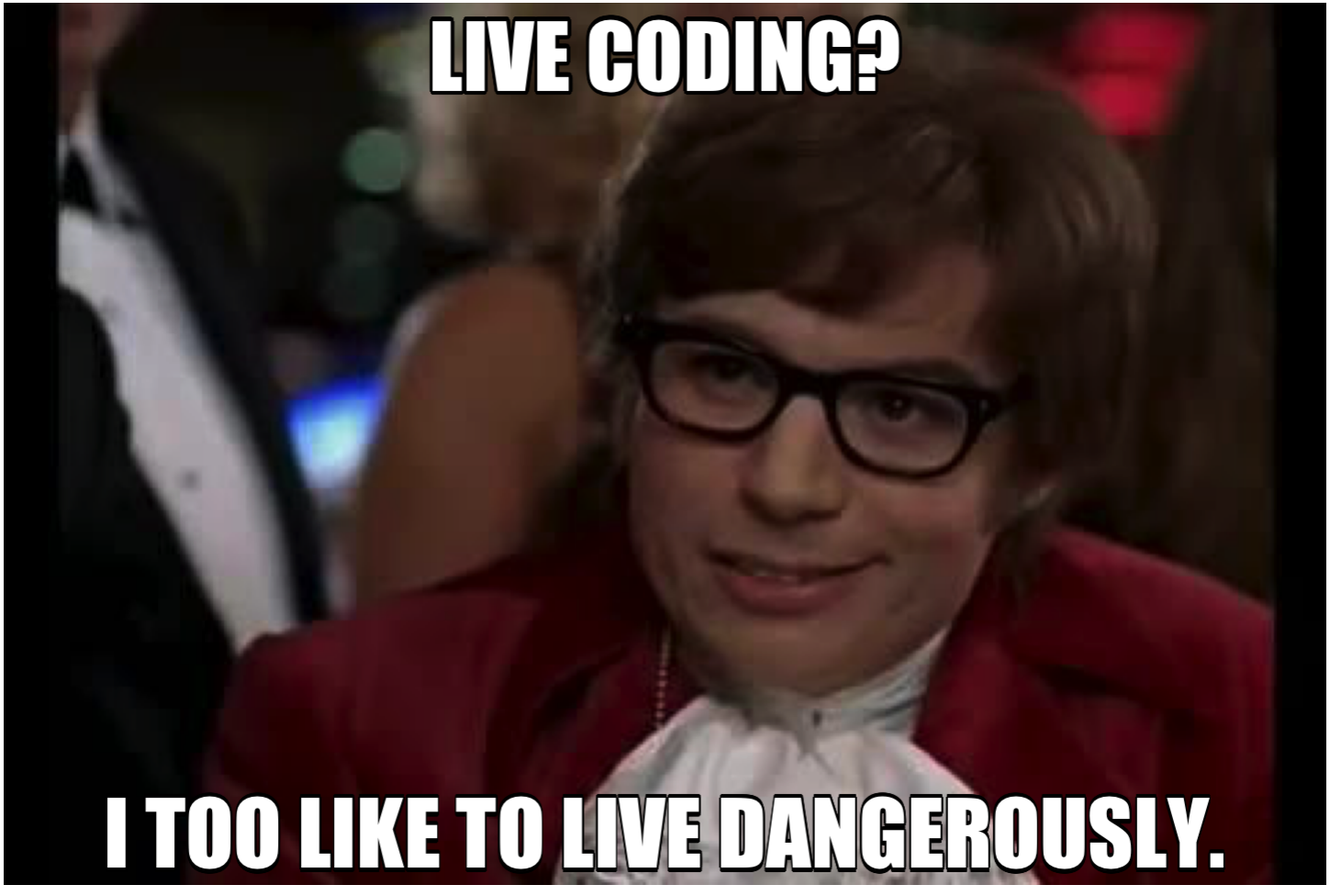
```
## code to prepare `dataset` dataset goes here

usethis::use_data("dataset")
```

Adding datasets

```
## code to prepare `dataset` dataset goes here
library(tidyverse)
my_app_dataset <- read.csv("bla/bla/bla.csv")
my_app_dataset <- my_app_dataset %>%
  filter(this == "that") %>%
  arrange(on_that) %>%
  select( -contains("this") )
usethis::use_data(my_app_dataset)
```

Now available as `my_app_dataset` inside your app.

Demo time 🎉



Your turn!

https://github.com/ColinFay/golem-joburg/tree/master/exo-part-2