

Shiny en production avec {golem}

Colin Fay - ThinkR

ThinkR x RStudio Roadshow, Paris



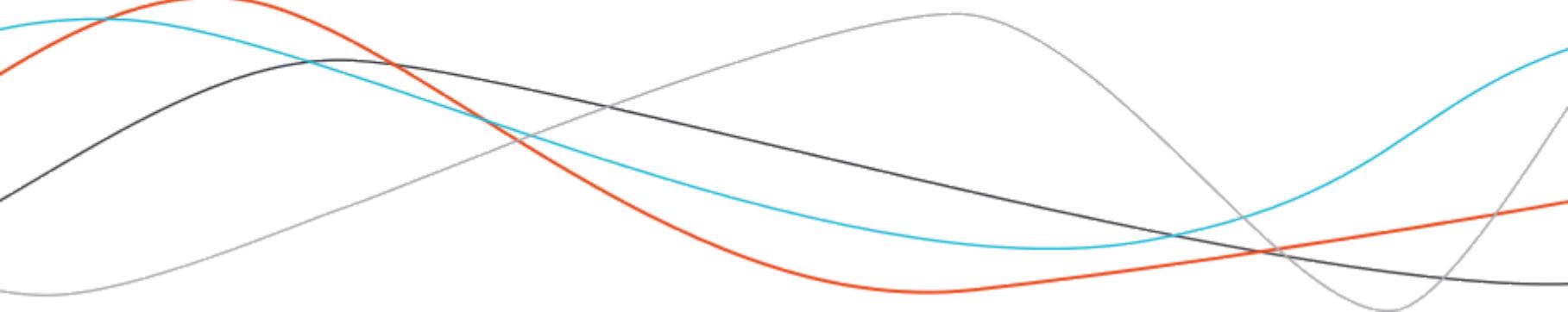
\$ whoami

Colin FAY

Data Scientist & R-Hacker at ThinkR

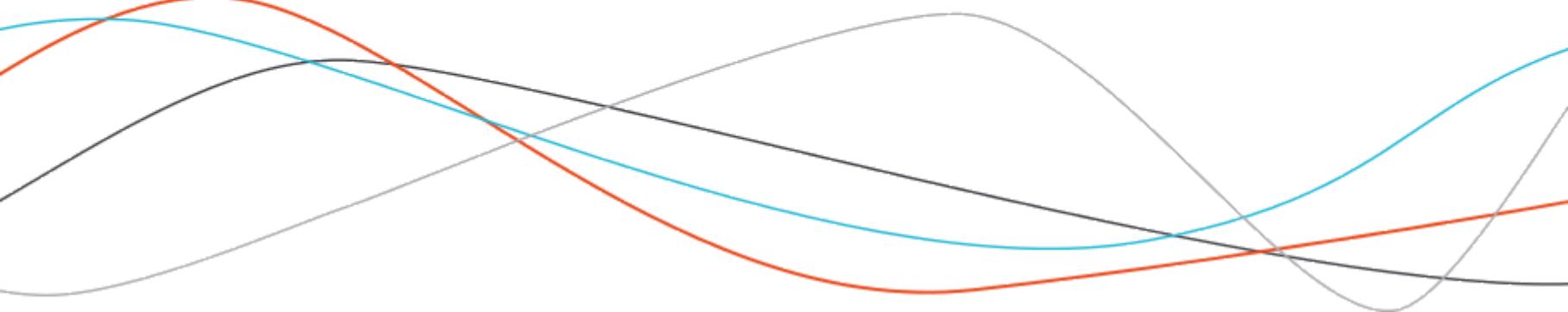
- <http://thinkr.fr>
- <http://rtask.thinkr.fr>
- http://twitter.com/thinkr_fr
- http://twitter.com/_colinfay
- <http://github.com/Thinkr-open>
- <http://github.com/colinfay>

>_ Mais avant de commencer : accéder aux serveurs RStudio





À propos de {golem}





{golem} ?

{golem} est un **package R** destiné à la création d'applications Shiny pour la production.

Utiliser ce package impose un cadre relativement strict, mais permet de se passer des réflexions techniques d'infrastructure. Même si le package semble imposant, en pratique le workflow est assez simple à suivre.

Installer {golem}

```
# install.packages("remotes")
remotes::install_github("Thinkr-open/golem")
```

Notes : il y a 1000 façon de faire une application Shiny, mais seulement une poignée de façon de le faire bien. {golem} est une approche qui a fait ses preuves en production.



Shiny App As a Package

Qu'est-ce qu'une app "prod-ready" ?

- Accompagnée de ses métadonnées (`DESCRIPTION`)
- Découpée en fonctions (`R/`)
- Testée (`tests/`)
- Avec les dépendances gérées (`NAMESPACE`)
- Documentée (`man/` & `vignettes`)

Donc, un 📦



Créer un {golem}

New Project

Back **Project Type**

- R Package using RcppEigen > ^
- R Package using RcppParallel >
-  Book Project using bookdown >
- R Package using devtools > ^
-  Package for Shiny App using golem > ^
-  New Plumber API Project >
- Simple R Markdown Website > ^

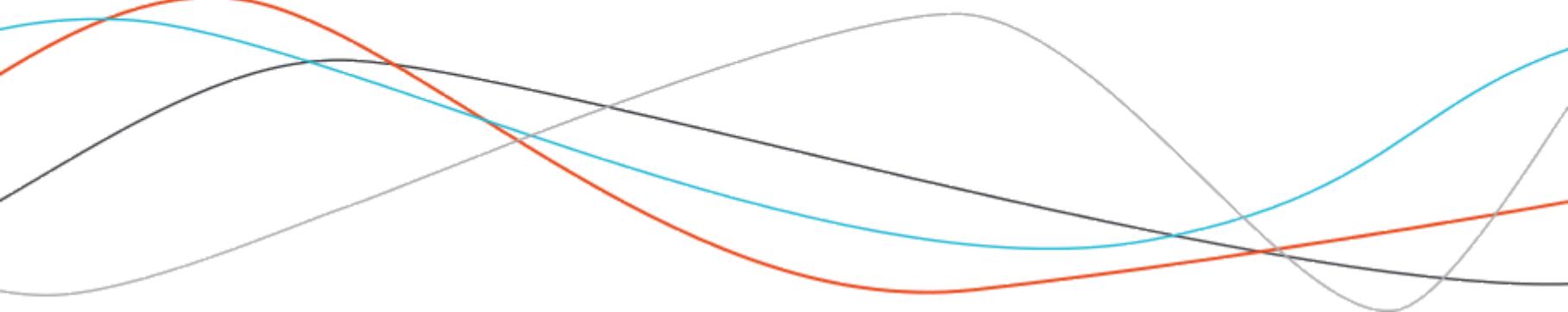
Create a new
Package for Shiny
App using golem

Cancel



À vous :

>_Créer votre premier {golem}.





Tester l'app en local

Pour lancer l'app en local, on va faire tourner le script dans `dev/run_dev.R`.

>_ Ouvrir `run_dev.R`, et lancer le script

>_ Rajouter un header dans `app_ui`

>_ Relancer `run_dev.R`



Déploiement sur Connect

```
golem::add_rconnect_file()  
usethis::use_build_ignore("app.R")
```

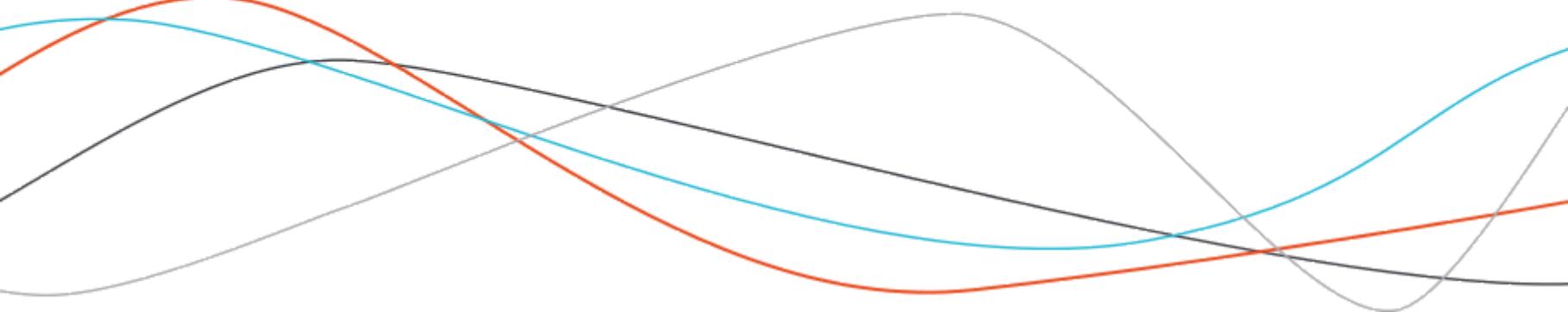
| Va créer le fichier `app.R` à la racine du package, qui contiendra :

```
# To deploy, run: rsconnect::deployApp()  
  
pkgload::load_all()  
options( "golem.app.prod" = TRUE)  
shiny::shinyApp(ui = app_ui(), server = app_server)
```

| On peut maintenant déployer le package sur Connect



Comprendre {golem}





DESCRIPTION

```
 |--dev/
    |--01_start.R
    |--02_dev.R
    |--03_deploy.R
    |--run_dev.R
 |--inst/
    |--app
        |--server.R
        |--ui.R
        |--www/
            |--favicon.ico
 |--man/
    |--run_app.Rd
NAMESPACE
monapp.Rproj
 |--R/
    |--app_server.R
    |--app_ui.R
    |--onload.R
    |--run_app.R
```

- DESCRIPTION & NAMESPACE : Méta données du package.
- dev/ : outils de dev.
- inst/app : on ajoutera des fichiers externes dans www/. On ne touche pas à ui.R et server.R.
- man : documentation de l'app, se génère automatiquement
- monapp.Rproj : projet RStudio.
- R/app_server.R, app_ui.R : ce sont les deux seuls fichiers que l'on va remplir.
- R/run_app.R & onload.R : fonction qui va lancer l'app, et la configurer.



01_start.R

Commençons par remplir les métadonnées (une fois pour toute) de notre package :

```
golem::fill_desc(  
  pkg_name = "nasapp",  
  pkg_title = "Visualisation en Direct de l'ISS",  
  pkg_description = "Visualisation en Direct la position de l'ISS",  
  author_first_name = "colin",  
  author_last_name = "fay" ,  
  author_email = "colin@thinkr.fr",  
  repo_url = NULL  
)
```



01_start.R

```
usethis::use_mit_license( name = "Colin Fay" )
usethis::use_readme_rmd()
usethis::use_code_of_conduct()
usethis::use_lifecycle_badge( "Experimental" )
usethis::use_news_md()

usethis::use_data_raw()

golem::use_recommended_tests()

golem::use_recommended_dep()

golem::use_utils_ui()
golem::use_utils_server()

golem::use_favicon( path = "path/to/favicon" )
```

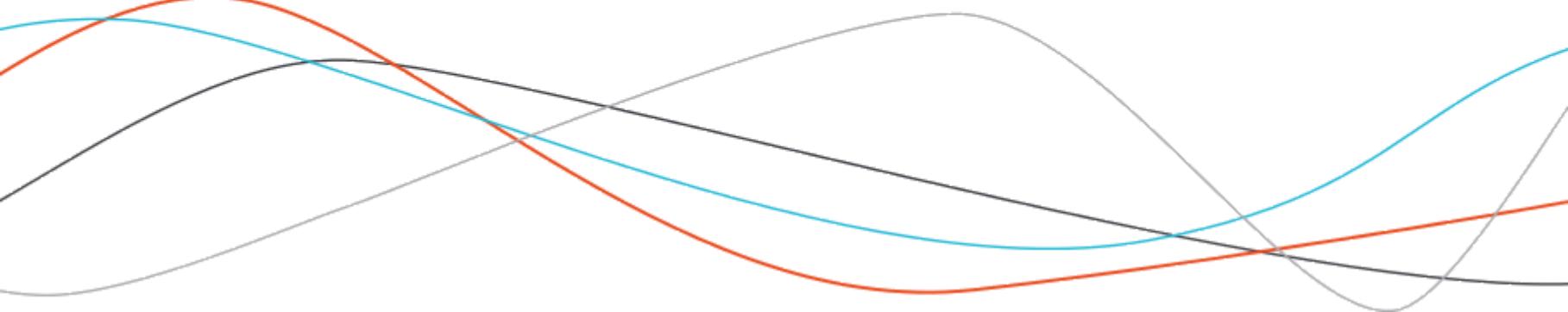


Créer des modules

Voilà, nous sommes prêts à générer des modules !

```
golem::add_module("premier_element")
```

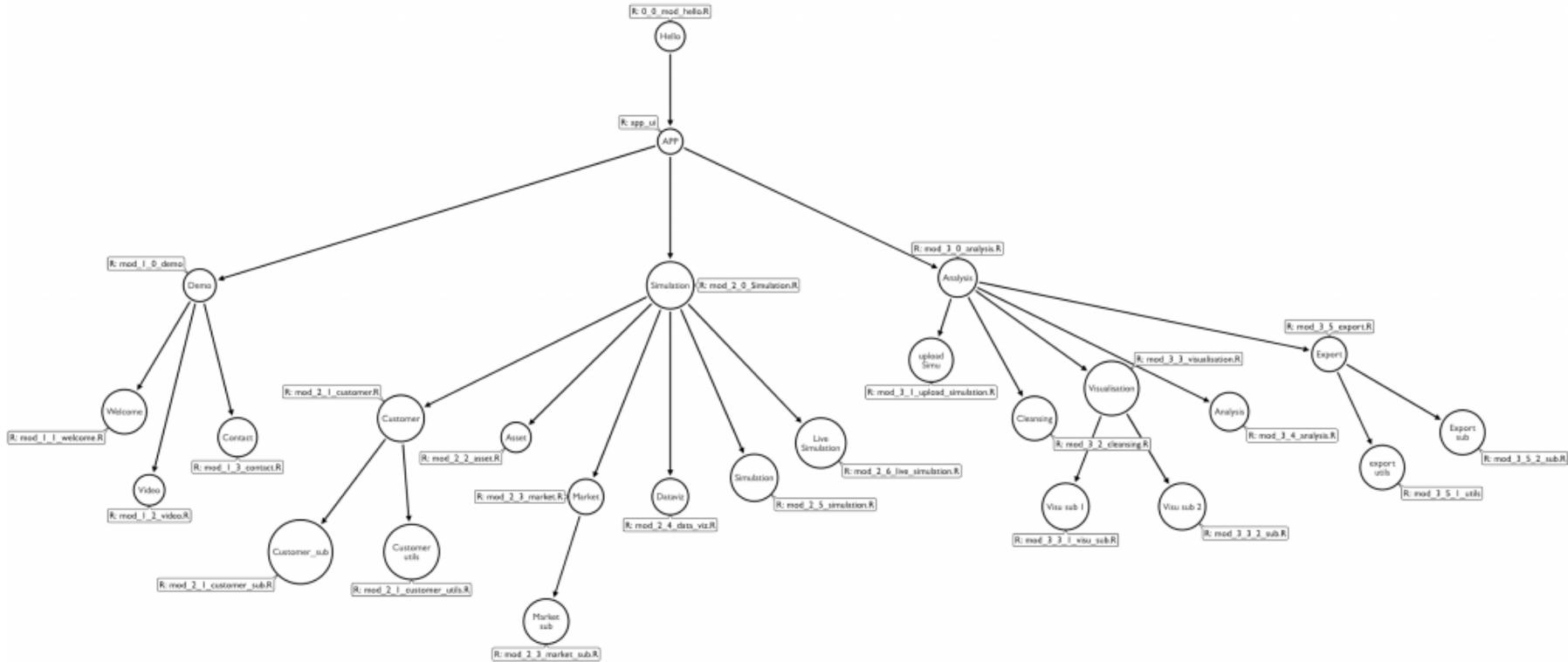
À propos des modules





C'est quoi un module ?

- Un module est un morceau d'application Shiny, "self contained", à intégrer dans un plus gros squelette.
- On les utilise pour découper l'application en sous éléments.
- Permet de ne plus avoir à gérer des applications de plusieurs milliers de lignes.
- Peut être réutilisé plusieurs fois.





Premier module

```
mod_premier_elementui <- function(id){  
  ns <- NS(id)  
  tagList(  
    # On met ici les inputs  
    # Tous les "id" d'input doivent se mettre dans `ns()`  
  )  
}  
  
mod_premier_element <- function(input, output, session){  
  ns <- session$ns  
  # On interprète ici les inputs, sans le ns()  
}
```



Mettre le premier module dans une app

```
app_ui <- function() {  
  fluidPage(  
    titlePanel("Old Faithful Geyser Data"),  
    mod_premier_elementui("premier_elementui_1")  
  )  
}  
  
app_server <- function(input, output, session) {  
  callModule(mod_premier_element, "premier_elementui_1")  
}
```



Exemple

```
mod_premier_element_ui <- function(id){  
  ns <- NS(id)  
  tagList(  
    selectInput(  
      ns("table"),  
      "Quelle table ?",  
      c("iris", "mtcars", "airquality")  
    ),  
    tableOutput(ns("out_table"))  
  )  
}
```



Exemple

```
mod_premier_element_server <- function(input, output, session) {  
  
  output$out_table <- renderTable({  
    head(get(input$table))  
  })  
  
}  
  
shinyApp(app_ui(), app_server)
```

Résumé

Séparation en modules :

- `app_ui` : fonction définissant l'interface utilisateur globale de l'application. Dans cette fonction globale, on trouve une série de modules d'UI, de la forme `mod_***ui("***ui_1")`.
- `app_server` : fonction définissant le serveur global de l'application. Dans cette fonction, une série de `callModule(mod_***, "mod_***ui_1")`.
- À chaque module sa combinaison UI/server.



À vous

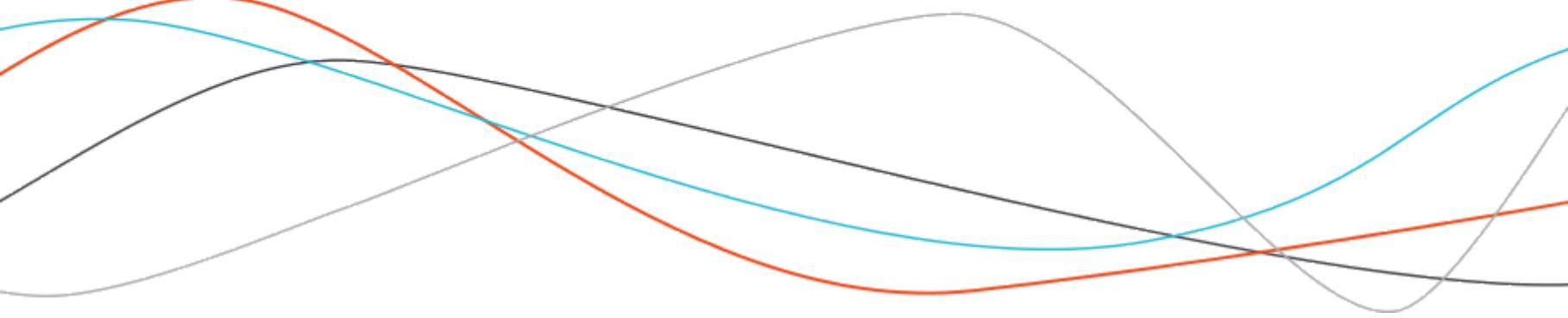
>_ Créer un module dans votre golem.



{golem} - workflow

- Lancer le projet
- Dans `dev/01_start.R`, remplir la description et lancer les fonctions de départ
- Lancer `dev/run_dev.R` pour s'assurer que tout va bien
- Fermer `dev/01_start.R`
- Dans `dev/02_dev.R`, créer un module avec `golem::add_module("plop")`.
- Copier coller `module_plop_ui("plop_ui_1")` dans `R/app_ui.R`
- Copier coller `callModule(module_plop_server, "plop_ui_1")` dans `R/app_server.R`
- Compléter le module
- Lancer `dev/run_dev.R` régulièrememnt pour s'assurer que tout va bien
- Créer un second module, et ainsi de suite...

Test, dépendances





02_dev.R

dépendances

```
usethis::use_package("pkg") # To call each time you need a new package
```

tests

```
usethis::use_test("app")
```



On test quoi ?

Dans notre golem, on va séparer les fonctions d'UI et les fonctions "backend".

- Fonctions de back-end : classiques, on va les tester comme des fonctions classiques, puisqu'elles ne sont pas censées être dépendantes de l'app.
- Fonctions de front-end : génèrent du HTML, et dépendent des inputs.

```
mod_premier_elementui <- function(id){  
  ns <- NS(id)  
  tagList(  
    sidebarLayout(  
      sidebarPanel(  
        sliderInput(ns("bins"),  
                   "Nombre de bins:",  
                   min = 1,  
                   max = 50,  
                   value = 30)  
      ),  
      mainPanel(  
        plotOutput("distPlot")  
      )  
    )  
  )  
}
```



On test quoi ?

```
mod_premier_elementui("plop")
```

```
<div class="row">
  <div class="col-sm-4">
    <form class="well">
      <div class="form-group shiny-input-container">
        <label class="control-label" for="plop-bins">Nombre de bins:</label>
        <input class="js-range-slider" id="plop-bins" data-min="1" data-
max="50" data-from="30" data-step="1" data-grid="true" data-grid-num="9.8"
data-grid-snap="false" data-prettify-separator="," data-prettify-
enabled="true" data-keyboard="true" data-data-type="number"/>
      </div>
    </form>
  </div>
  <div class="col-sm-8">
    <div id="distPlot" class="shiny-plot-output" style="width: 100% ;
height: 400px"></div>
  </div>
</div>
```



On test quoi ?

En amont :

```
htmltools::save_html(mod_premier_elementui("plop"), "ui.html")
```

Dans les tests:

```
test_that("ui de mon premier element", {  
  premier_el <- mod_premier_elementui("plop")  
  golem::expect_html_equal(premier_el, "ui.html")  
})
```



On test quoi ?

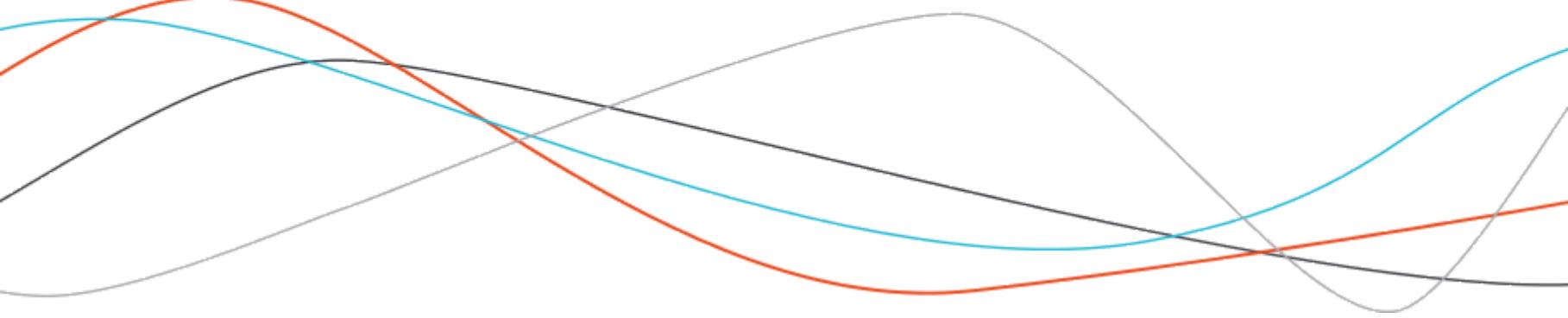
Est-ce que l'appli se lance ?

À noter qu'il faudra configurer différemment sur vos serveur de CI (Gitlab, travis...)

```
context("launch")
library(processx)
testthat::test_that(
  "app launches", {
    # On lance l'app
    x <- process$new(
      "R",
      c( "-e", "setwd('..../'); pkgload::load_all();run_app()" )
    )
    # On laisse l'app se charger
    Sys.sleep(5)
    # On vérifie que le process est alive
    expect_true(x$is_alive())
    x$kill()
  }
}
```



JS & CSS





Templates CSS et JS

```
golem::add_js_file("script")
golem::add_js_handler("script")
golem::add_css_file("custom")
```

■ Pas besoin de spécifier les extensions

On les liste ensuite dans `golem_add_external_resources()` de `app_ui()`.

■ On y accède avec, par exemple, `tags$img(src = "www/pics.jpeg")`



golem::js()

golem vient avec une série de fonctions JavaScript que vous pouvez appeler depuis le serveur.

Ces fonctions sont mises par défaut dans l'app_ui avec golem::js().

On les appelle avec session\$sendCustomMessage("fonction", "reference_ui").

Ces fonctions prennent toutes une "reference_ui", définissant l'élément de l'UI avec lequel on va interagir. Cette référence peut être un "selecteur jQuery" (pour de la référence flexible), d'autres un id, d'autres une classe.



golem::js()

- `showid` et `hideid`, `showclass` et `hideclass` montrent ou cache des éléments avec l'id ou la classe définie.
- `showhref` et `hidehref` : même chose, mais tente de matcher un lien `href`
- `clickon` clique sur un élément. Le selecteur jQuery complet doit être utilisé.
- `show` et `hide` cachent et montre des éléments sélectionné avec le selecteur jQuery complet.



Quelques sélecteur jQuery

- `#plop` : l'élément à l'ID `plop`
- `.pouet` : les éléments de class `pouet`
- `"button:contains('Afficher')"` : les boutons dont le texte contient "Afficher".

En html, les éléments sont caractérisés par des attributs :

```
<a href = "https://thinkr.fr" data-value = "panel2">ThinkR</a>
```

Contient les attributs `href` et `data-value`. On peut y référer entre `[]` après le nom du tag :

- `a[href = "https://thinkr.fr"]` : le lien dont l'élément `href` est `https://thinkr.fr`
- `a[data-value="panel2"]` : le lien dont l'attribut `data-value` est `"panel2"`

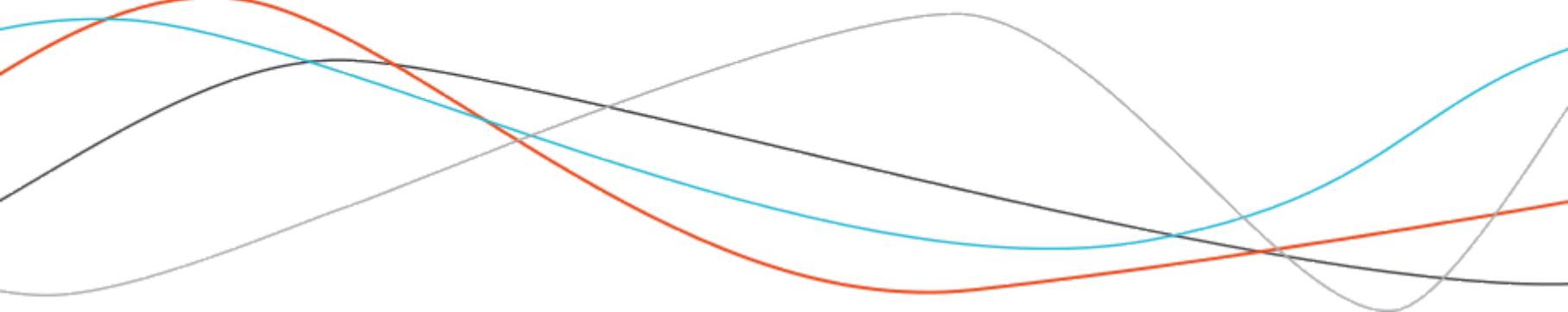


Documentation

```
# Vignette  
usethis::use_vignette("bing")  
devtools::build_vignettes()
```

```
# Code coverage  
usethis::use_travis()  
usethis::use_appveyor()  
usethis::use_coverage()
```

Déploiement sur Connect





Déploiement

En supposant que nous ayons déjà fait le déploiement une fois à la main.

```
library(attempt)

secure_send_connect <- function( where = "." ) {
  res <- devtools::check(where)
  stop_if_not(res$errors, ~length(.x) == 0, "Erreur, app non déployée")
  rsconnect::deployApp(forceUpdate = TRUE)
}

secure_send_connect()
```



Thx! Questions?

Colin Fay

colin@thinkr.fr

<https://thinkr.fr/>

http://twitter.com/_colinfay

<https://rtask.thinkr.fr/>

http://twitter.com/thinkr_fr

<https://colinfay.me/>

<https://github.com/ColinFay>