



CESAB
CENTRE DE SYNTHÈSE ET D'ANALYSE
SUR LA BIODIVERSITÉ



Building an R package

the best part 😊



Nicolas CASAJUS

{ Data scientist FRB-CESAB }

Mardi 3 novembre 2020

Everything's going to be okay



What's an R Package?

“ In R, the fundamental unit of shareable code is the package. A package bundles together `code`, `data`, `documentation`, and `tests`, and is easy to share with others

— Hadley Wickham

- An R package is a collection of **well-documented functions**
- It makes your work more **reproducible**
- It makes your code **useful** for (you and) others

What's an R Package?

“ In R, the fundamental unit of shareable code is the package. A package bundles together **code**, **data**, **documentation**, and **tests**, and is easy to share with others

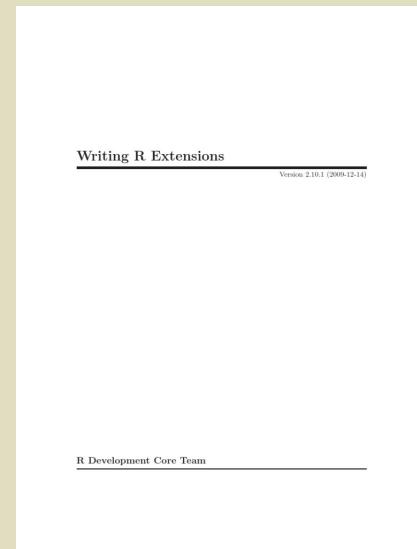
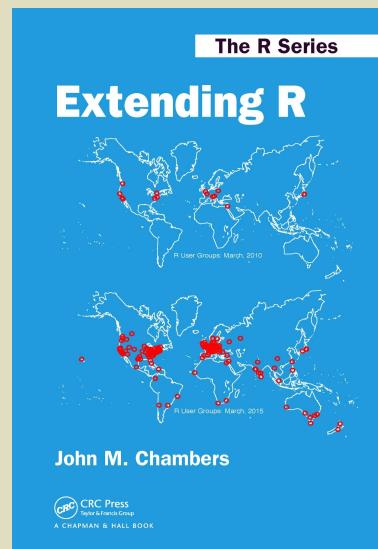
— Hadley Wickham

- An R package is a collection of **well-documented functions**
- It makes your work more **reproducible**
- It makes your code **useful** for (you and) others
- It's a lot of **fun!**

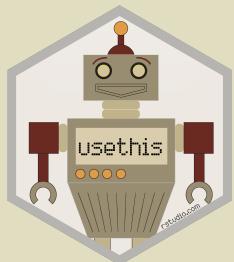
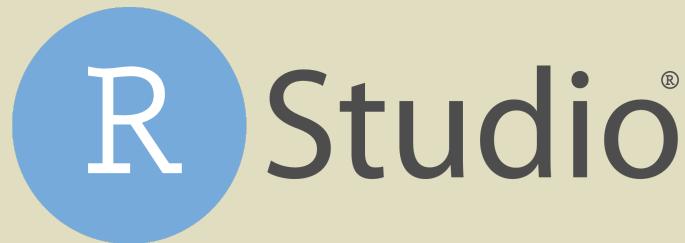


What's an R Package?

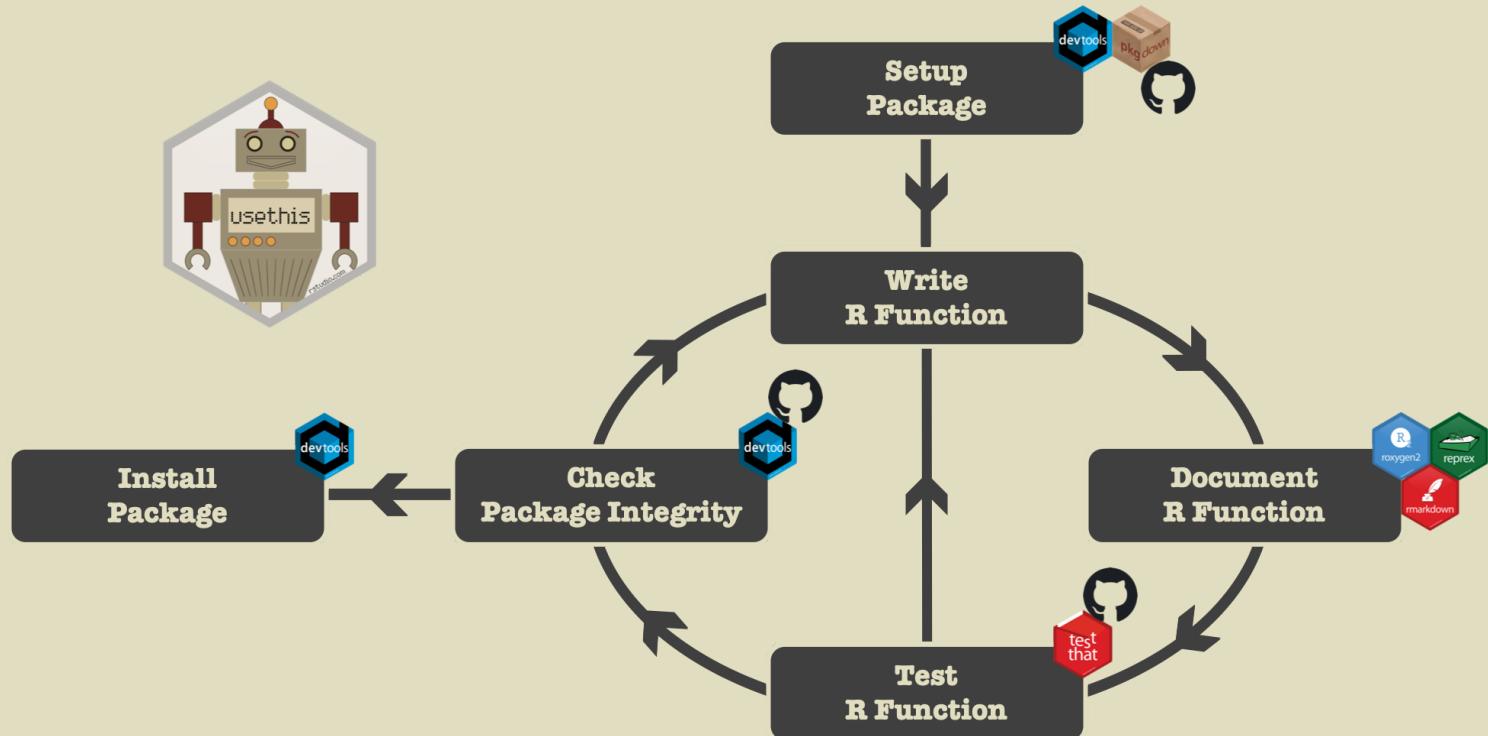
- As of today (2020-11-08), **16510** packages are available on the **CRAN**. And many many more on GitHub and Bioconductor!
- Must-read resources:



Recommended environment



Development workflow



Live coding session



Session information

```
devtools::session_info()
```

— Session info —

```
Version R version 4.0.3 (2020-10-10)
OS      macOS Catalina 10.15.7
System  x86_64, darwin17.0
UI      RStudio
Language English
Collate fr_FR.UTF-8
CTYPE   fr_FR.UTF-8
TZ      Europe/Paris
Date    2020-11-03
```

— Packages —

| Package | Version | Date | Source |
|-----------|---------|------------|----------------|
| devtools | 2.3.2 | 2020-09-18 | CRAN (R 4.0.2) |
| knitr | 1.30.0 | 2020-09-22 | CRAN (R 4.0.2) |
| pkgdown | 1.6.1 | 2020-09-12 | CRAN (R 4.0.2) |
| remotes | 2.2.0 | 2020-07-21 | CRAN (R 4.0.2) |
| rmarkdown | 2.5.0 | 2020-10-21 | CRAN (R 4.0.3) |
| roxygen2 | 7.1.1 | 2020-06-27 | CRAN (R 4.0.2) |
| testthat | 2.3.2 | 2020-03-02 | CRAN (R 4.0.0) |
| usethis | 1.6.3 | 2020-09-17 | CRAN (R 4.0.2) |

Package skeleton



Create the structure



New Project Wizard

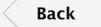
Create Project

-  **New Directory**
Start a project in a brand new working directory >
-  **Existing Directory**
Associate a project with an existing working directory >
-  **Version Control**
Checkout a project from a version control repository >

Cancel

New Project Wizard

Project Type

-  Back
- R Package using RcppArmadillo >
- R Package using RcppEigen >
- R Package using RcppParallel >
-  Website using blogdown >
-  Book Project using bookdown >
- R Package using devtools** >
- Simple R Markdown Website >

Cancel

👉 Make sure to select **R Package using devtools**

⚠ A package name can **only contain** letters and numbers (the dot is the only non-alphanumeric character allowed)

Create the structure

- Using  directly (no IDE)

```
usethis::create_package("/Users/nicolascasajus/Desktop/rpkg")
```

- 👉 Make sure to specify the **absolute path** to your package

```
✓ Creating '/Users/nicolascasajus/Desktop/rpkg'  
✓ Setting active project to '/Users/nicolascasajus/Desktop/rpkg'  
✓ Creating 'R/'  
✓ Writing 'DESCRIPTION'  
Package: rpkg  
Title: What the Package Does (One Line, Title Case)  
Version: 0.0.0.9000  
Authors@R (parsed):  
  * First Last <first.last@example.com> [aut, cre] (YOUR-ORCID-ID)  
Description: What the package does (one paragraph).  
License: `use_mit_license()`, `use_gpl3_license()` or friends  
Encoding: UTF-8  
LazyData: true  
Roxygen: list(markdown = TRUE)  
RoxygenNote: 7.1.1  
✓ Writing 'NAMESPACE'  
✓ Changing working directory to '/Users/nicolascasajus/Desktop/rpkg'
```

Create the structure

- Let's take a look at the package structure

The screenshot shows the RStudio interface with the title bar "rpkg - RStudio". The left sidebar displays the file structure of the package:

- ..
- .gitignore
- .Rbuildignore
- DESCRIPTION
- NAMESPACE
- R
- rpkg.Rproj

The "rpkg.Rproj" file is highlighted with a red box. The main console area shows the R startup message:

```
R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak Out"  
Copyright (C) 2020 The R Foundation for Statistical Computing  
Platform: x86_64-apple-darwin17.0 (64-bit)  
  
R est un logiciel libre livré sans AUCUNE GARANTIE.  
Vous pouvez le redistribuer sous certaines conditions.  
Tapez 'license()' ou 'licence()' pour plus de détails.  
  
R est un projet collaboratif avec de nombreux contributeurs.  
Tapez 'contributors()' pour plus d'information et  
'citation()' pour la façon de le citer dans les publications.  
  
Tapez 'demo()' pour des démonstrations, 'help()' pour l'aide  
en ligne ou 'help.start()' pour obtenir l'aide au format HTML.  
Tapez 'q()' pour quitter R.
```

The status bar at the bottom right shows "rpkg" with a dropdown arrow.

! What about reproducibility?

- We will create an **R** script with all the command lines required to develop our package so we can reuse it in the future for new developments.
- This file will be named **_devhistory.R** and will be saved **at the root** of the project (and not in the **R/** folder).

```
usethis::edit_file("_devhistory.R")
```

👉 From now, **all the command lines** will be written in this file.

⚠ As this file is not part of a typical package structure, we need to tell R to ignore it when checking and installing the package. It's the purpose of the **.Rbuildignore** file.

```
usethis::use_build_ignore("_devhistory.R")
```

✓ Adding '^_devhistory\\.R\$' to '.Rbuildignore'

Setup versioning

- Let's initialize the **git** versioning

```
usethis::use_git(message = ":tada: Initial commit")
```

```
✓ Setting active project to '/Users/nicolascasajus/Desktop/rpkg'  
✓ Initialising Git repo  
✓ Adding '.Rhistory', '.RData' to '.gitignore'
```

There are 6 uncommitted files:

```
'_devhistory.R'  
' .gitignore'  
' .Rbuildignore'  
'DESCRIPTION'  
'NAMESPACE'  
' rpkg.Rproj'
```

Is it ok to commit them?

(...)

- A restart of RStudio is required to activate the Git pane

Restart now?

(...)

👉 Commit changes and restart RStudio

.DS_Store files

.DS_Store files are hidden files storing folder preferences on macOS computers. It's recommended to remove them from the versioning. Moreover these files are not part of standard package structure: we also need to ignore them during the check and installation of the package.

- Let's add it to .gitignore and .Rbuildignore files

```
usethis::use_git_ignore(".DS_Store")
usethis::use_build_ignore(".DS_Store")
```

- ✓ Adding '.DS_Store' to '.gitignore'
- ✓ Adding '^\\.DS_Store\$' to '.Rbuildignore'

- And let's commit changes

```
usethis::use_git(message = ":see_no_evil: Ban .DS_Store files")
```

- ✓ Adding files
- ✓ Commit with message ':see_no_evil: Ban .DS_Store files'

Package metadata



Package metadata

- Before we go any further, we will edit some informations about our package using the **DESCRIPTION** file

```
usethis::edit_file("DESCRIPTION")
```

```
Package: rpkg
Title: What The Package Does (one line, title case required, no final period)
Authors@R:
  person(given = "First",
         family = "Last",
         role   = c("aut", "cre"),
         email  = "first.last@example.com",
         comment = c(ORCID = "YOUR-ORCID-ID"))
Description: What the package does (one paragraph). The description of a package
             is usually long, spanning multiple lines. The second and subsequent lines
             should be indented, usually with four spaces.
```

Package metadata

- Before we go any further, we will edit some informations about our package using the **DESCRIPTION** file

```
usethis::edit_file("DESCRIPTION")
```

```
Package: rpkg
Title: A Minimal But Complete R Package
Authors@R:
  person(given = "Nicolas",
         family = "Casajus",
         role   = c("aut", "cre"),
         email  = "nicolas.casajus@fondationbiodiversite.fr",
         comment = c(ORCID = "0000-0002-5537-5294"))
Description: The purpose of the \code{rpkg} package is to illustrate the main
             structure and components of an R Package with respect to the CRAN submission
             policies (<https://cran.r-project.org/>).
```

⚠ The other fields will be added/modified automatically.

👉 Resources: **R Package - Chap8**

```
usethis::use_git(message = ":bulb: Edit package metadata")
```

Package-level documentation

- What about converting the **DESCRIPTION** informations into a package-level documentation file? This file will be the homepage of the help section of our package accessible with `?pkg_name` or `??pkg_name`

```
usethis::use_package_doc()
```

✓ Writing 'R/rpkg-package.R'

⚠ A dummy file has been created in R/: **do not edit this file!**

- Now we need to generate the corresponding documentation file (**.Rd**)

```
devtools::document()
```

✓ Writing 'NAMESPACE'

✓ Writing 'man/rpkg-package.Rd'

```
usethis::use_git(message = ":bulb: Update documentation")
```

Package license



Package license

- Choosing a license is important if you're planning on releasing your package
- There are two families of open-source licenses: **permissive** (e.g. MIT, Apache) and **copyleft** (e.g. GPL)

👉 Resources: choosealicense.com

- Let's use the **MIT** license, a permissive one

```
usethis::use_mit_license(name = "Nicolas Casajus")
```

```
✓ Setting License field in DESCRIPTION to 'MIT + file LICENSE'  
✓ Writing 'LICENSE.md'  
✓ Adding '^LICENSE\\\\.md$' to '.Rbuildignore'  
✓ Writing 'LICENSE'
```

```
usethis::use_git(message = ":page_facing_up: Add package license")
```

Function implementation



A first R function

- Let's create a function called `moyenne()`, an equivalent of the R function `mean()`

A first R function

- Let's create a function called `moyenne()`, an equivalent of the R function `mean()`
- We'll use `usethis::use_r()` to create a file with the same name*

```
usethis::use_r("moyenne")
```

- Modify 'R/moyenne.R'
- Call `use_test()` to create a matching test file

- Let's implement the core of the function

```
moyenne <- function(x) sum(x) / length(x)
```

👉 Resources: **Tidyverse style guide** and the R package `{styler}` (with its RStudio addin) to format your R code

* You can also have several functions in one single R file (grouped by theme)

Let's try our function

👉 Before going any further we have to try our code. So we will load our package (and **NOT** sourcing the function)

```
devtools::load_all()
```

Now we can use our function

```
moyenne(1:10)
```

```
## [1] 5.5
```

Let's compare with the function `mean()`

```
mean(1:10)
```

```
## [1] 5.5
```

🎉🎉🎉 Yeah!!! 🎉🎉🎉

```
usethis::use_git(message = ":boom: New feature - moyenne()")
```

Time to document



Time to document



- Specially-structured comments **preceding** each function definition
- Lightweight syntax easy to write and to read
- Syntax: `#' @field value`
- Keep function definition and documentation in the same file
- Automatically write `.Rd` files and **NAMESPACE**

Each **Roxygen2 header** will always start with these two fields:

```
#' @title Short Title of the Function (One Line)
#'
#' @description A longer description of what the function does (several lines)
```

👉 Keywords `@title` and `@description` can be omitted

```
#' Short Title of the Function (One Line)
#'
#' A longer description of what the function does (several lines)
```

Time to document

- 👉 If your function has **parameters**, each one must be documented

```
#' @param param_name param_description
```

For example with our function `moyenne()`

```
#' @param x a numerical vector
```

- 👉 If your function **returns** an R object use the keyword `@return`

```
#' @return What the function returns
```

For example with our function `moyenne()`

```
#' @return The arithmetic mean of the values as a numeric vector of length one.
```

You can omit this field if your function returns nothing

Time to document

- 👉 Add a section `@examples` to show how to use your function

```
#' @examples  
#' x <- 1:10  
#' moyenne(x)
```

If you don't want your example to be executed use `\dontrun{}` (in case your example returns an error or in case of time consuming code)

```
#' @examples  
#' \dontrun{  
#' x <- 1:10  
#' moyenne(x)  
#' }
```

- 👉 You can also add a reproducible example (dataset added to your package)

Time to document

- 👉 Finally if you want your function to be used directly by user you need to add this tag

```
#' @export
```

- 👉 Additional Roxygen2 tags

```
#' @details  
#' @author  
#' @references  
#' @seealso  
#' @keywords  
#' @section  
#' @alias  
#' @family
```

- 👉 Resources: [R Package - Chap10](#) and [R Package Primer](#)

Time to document

```
#' Arithmetic Mean
#'
#' This function computes the arithmetic mean of a numerical vector.
#'
#' @param x a numerical vector
#'
#' @return The arithmetic mean of the values as a numeric vector of length one.
#'
#' @export
#'
#' @examples
#' x <- 1:10
#' moyenne(x)

moyenne <- function(x) sum(x) / length(x)
```

Time to document

👉 It's time to generate the corresponding `.Rd` file (special file for R documentation) from this Roxygen2 header

```
devtools::document() # or roxygen2::roxygenize()
```

✓ Writing 'NAMESPACE'
✓ Writing 'man/moyenne.Rd'

- In addition to the creation of `man/moyenne.Rd` file, the `NAMESPACE` has been updated

```
# Generated by roxygen2: do not edit by hand
export(moyenne)
```

👉 As we can see this file lists which functions need to be exported (accessible by the user when loading the package). But it also deals with external dependencies.

```
usethis::use_git(message = ":bulb: Update documentation")
```

Checking package



Checking package

👉 It's time to check the integrity of our package

```
devtools::check()
```

— R CMD check —

- ✓ checking package namespace information
- ✓ checking package dependencies (1.9s)
- ✓ checking DESCRIPTION meta-information ...
- ✓ checking index information
- ✓ checking package subdirectories ...
- ✓ checking R files for syntax errors ...
- ✓ checking dependencies in R code ...
- ✓ checking R code for possible problems (1.8s)
- ✓ checking Rd files ...
- ✓ checking Rd metadata ...
- ✓ checking Rd contents ...
- ✓ checking examples (526ms)

— R CMD check results —

rpkg 0.0.0.9000

Duration: 10.8s

0 errors ✓ | 0 warnings ✓ | 0 notes ✓

🎉🎉🎉 Yeah!!! 🎉🎉🎉

Installing package

👉 Finally we can install our package

```
devtools::install()
```

```
DONE (rpkg)
```

```
library("rpkg")
moyenne(1:1000)
```

```
## [1] 500.5
```

Or,

```
rpkg::moyenne(1:1000)
```

Back to tests



gifbin.com

Test your code

- Testing is a vital part of package development
- But until now we just tried our code informally and on the fly
- Problem: it's time consuming, repetitive and it can break the code

👉 Package `{testthat}`



- Implements a lot of unit tests
- Formal automated testing
- Explicits how your code should behave
- Makes your code more robust

Test your code

- Testing is a vital part of package development
- But until now we just tried our code informally and on the fly
- Problem: it's time consuming, repetitive and it can break the code

👉 Package `{testthat}`



- Implements a lot of unit tests
- Formal automated testing
- Explicits how your code should behave
- Makes your code more robust

Let's add `{testthat}` to our package

```
usethis::use_testthat()
```

- ✓ Adding 'testthat' to Suggests field in DESCRIPTION
- ✓ Creating 'tests/testthat/'
- ✓ Writing 'tests/testthat.R'
- Call `use_test()` to initialize a basic test file and open it for editing.

Test your code

- 👉 A new  file has been written: `tests/testthat.R`. It sets the testing environment

```
library(testthat)
library(rpkg)
test_check("rpkg")
```

```
usethis::use_git(message = ":white_check_mark: Setup testthat")
```

- Now we have to implement some unit tests. Let's create an  script for testing the function `moyenne()`

```
usethis::use_test("moyenne")
```

- ✓ Writing 'tests/testthat/test-moyenne.R'
- Modify 'tests/testthat/test-moyenne.R'

- 👉 All tests files are stored in `tests/testthat/` and their names must start with `test-*`

Test structure

Tests are organised hierarchically: **expectations** are grouped into **tests** which are organised in **files**

- An **expectation** is the atom of a test. It describes the expected result of a computation. Expectations are functions that start with `expect_*`.
- A **test** has multiple expectations and tests one unit of a functionality (output, parameters, etc.). A test is created with `test_that()`.
- A test file groups multiple tests and has a **context** providing a short description of its tests. The context is specified by `context()`.

👉 Let's add some tests in `tests/testthat/test-moyenne.R`

```
context("Testing moyenne()")

test_that("check outputs", {
  expect_length(moyenne(1:3), 1)                         # Length of the output
  expect_equal(moyenne(1:3), 2)                            # Value of the output
})
```

Run the test

👉 To run the test we will use `devtools::test()`

```
devtools::test()
```

```
✓ |  OK F W S | Context
✓ |    2        | Testing moyenne()
```

```
== Results ==
```

```
OK:    2
Failed: 0
Warnings: 0
Skipped: 0
```

Test your code

Let's add another test

```
context("Testing moyenne()")  
  
test_that("check outputs", {  
  expect_length(moyenne(1:3), 1) # Length of the output  
  expect_equal(moyenne(1:3), 2) # Value of the output  
  expect_equal(moyenne(c(1:3, NA)), 2) # Check for NA  
})
```

```
devtools::test()
```

| | OK | F | W | S | Context |
|---|----|---|---|---|-------------------|
| x | 2 | 1 | | | Testing moyenne() |

```
test-moyenne.R:8: failure: check outputs  
moyenne(c(1:3, NA)) not equal to 2.
```

```
== Results ==  
OK: 2  
Failed: 1  
Warnings: 0  
Skipped: 0
```

Hum...

Our function does not seem to work as expected.

👉 We need to change the code of the function `moyenne()` to deal with `NA` values.

```
moyenne <- function(x) {  
  x <- na.omit(x)  
  sum(x) / length(x)  
}
```

Let's test again

```
devtools::test()
```

```
✓ |  OK F W S | Context  
✓ |    3       | Testing moyenne()
```

===== Results =====

```
OK:      3  
Failed:  0  
Warnings: 0  
Skipped: 0
```

That's better, but...

This is not a good practice. If user has **NA** values, this implementation will not inform him and makes the decision to remove **NA**. Instead we are going to let user choose to delete the **NA** or not.

👉 Let's add an additional parameter to our function: **na_rm** with a default value (**FALSE**). If **x** contains **NA** values and **na_rm = FALSE**, then an error will be returned. Otherwise (**na_rm = TRUE**) **NA** values will be removed and the computation can be done.

```
moyenne <- function(x, na_rm = FALSE) {  
  if (any(is.na(x))) {  
    if (na_rm) {  
      x <- na.omit(x)  
    } else {  
      stop("x contains NA values. Use na_rm = TRUE.")  
    }  
  }  
  sum(x) / length(x)  
}
```

That's better, but...

Let's modify tests

```
context("Testing moyenne()")  
  
test_that("check outputs", {  
  expect_length(moyenne(1:3), 1)  
  expect_equal(moyenne(1:3), 2)  
  expect_error(moyenne(c(1:3, NA)))  
  expect_equal(moyenne(c(1:3, NA), na_rm = TRUE), 2)  
})
```

And test again

```
devtools:::test()  
  
✓ |  OK F W S | Context  
✓ |    4        | Testing moyenne()  
  
===== Results =====  
OK:    4  
Failed: 0  
Warnings: 0  
Skipped: 0
```

and so on... to finally...



Final R function

```
moyenne <- function(x, na_rm = FALSE) {  
  
  # Check 'x' argument ----  
  if (missing(x)) {  
    stop("Missing x.")  
  }  
  if (is.null(x)) {  
    stop("x must be a numerical vector.")  
  }  
  if (sum(is.na(x)) == length(x)) {  
    stop("x cannot be NA.")  
  }  
  if (!is.vector(x)) {  
    stop("x must be a numerical vector.")  
  }  
  if (!is.numeric(x)) {  
    stop("x must be a numerical vector.")  
  }  
  if (length(x) < 2) {  
    stop("x must be length > 1.")  
  }  
  
  # Check 'na_rm' argument [...] ----  
  # Remove NA (if required) [...] ----  
  # Compute mean [...] ----  
}
```

Final R function

```
moyenne <- function(x, na_rm = FALSE) {  
  # Check 'x' argument [...] ----  
  
  # Check 'na_rm' argument ----  
  if (is.null(na_rm)) {  
    stop("na_rm must be TRUE or FALSE.")  
  }  
  if (sum(is.na(na_rm)) == length(na_rm)) {  
    stop("na_rm cannot be NA.")  
  }  
  if (length(na_rm) > 1) {  
    stop("na_rm must be TRUE or FALSE.")  
  }  
  if (!is.logical(na_rm)) {  
    stop("na_rm must be TRUE or FALSE.")  
  }  
  
  # Remove NA (if required) [...] ----  
  # Compute mean [...] ----  
}
```

Final R function

```
moyenne <- function(x, na_rm = FALSE) {  
  # Check 'x' argument [...] ----  
  # Check 'na_rm' argument [...] ----  
  
  # Remove NA (if required) ----  
  if (any(is.na(x))) {  
    if (na_rm) {  
      x <- na.omit(x)  
      if (length(x) < 2) {  
        stop("x has < 2 non-NA values.")  
      }  
    } else {  
      stop("x contains NA values. Use na_rm = TRUE.")  
    }  
  }  
  
  # Compute mean ----  
  sum(x) / length(x)  
}
```

Update documentation

```
#' Arithmetic Mean
#'
#' This function computes the arithmetic mean of a numerical vector.
#'
#' @param x a numerical vector (can contain 'NA' values).
#' @param na_rm a logical value indicating whether 'NA' values should be
#' stripped before the computation proceeds.
#'
#' @return The arithmetic mean of the values as a numeric vector of length one.
#'
#' @details An error will be returned if 'x' contains 'NA' values and 'na_rm' is
#' set to 'FALSE' or if 'x' contains less than two values (after removing 'NA').
#'
#' @export
#'
#' @examples
#' \dontrun{
#' moyenne(c(1:10, NA)) # error
#' }
#' moyenne(c(1:10, NA), na_rm = TRUE)

moyenne <- function(x, na_rm = FALSE) { ... }
```

```
devtools::document()
```

Run new tests

```
devtools::test()
```

```
✓ |  OK F W S | Context
✓ |  21        | Testing moyenne() [0.1 s]
```

```
== Results =
```

```
Duration: 0.1 s
```

```
OK:      21
Failed:   0
Warnings: 0
Skipped:  0
```

👉 Complete tests file available [here](#)

Check package integrity

```
devtools::check()
```

```
— R CMD check results ————— rpkg 0.0.0.9000 —————
Duration: 11.6s

> checking R code for possible problems ... NOTE
moyenne: no visible global function definition for 'na.omit'
Undefined global functions or variables:
  na.omit
Consider adding
  importFrom("stats", "na.omit")
to your NAMESPACE file.

0 errors ✓ | 0 warnings ✓ | 1 note x
```

👉 **One note:** Let's talk about package dependencies

Package dependencies



NAMESPACE

- 👉 Usually a package depends on functions of others packages. So we need to tell  about that to ensure that 1) others packages are installed when your package is installed, and 2) that they are made available when your package is loaded*.
- 👉 It's the purpose of the **NAMESPACE**
- 👉 As said before, the **NAMESPACE** is automatically updated when `devtools::document()` is called. So we need to list these dependencies in the **Roxygen2** header with:
 - `#' @import package`: will import all the functions of the dependency
 - `#' @importFrom package function`: will only import the listed functions

*This is true for every package (except for the package `{base}`), even for packages `{utils}`, `{stats}` and `{graphics}` installed by default with .

NAMESPACE

Previously we had the following note:

```
> checking R code for possible problems ... NOTE  
moyenne: no visible global function definition for 'na.omit'  
Undefined global functions or variables:  
  na.omit  
Consider adding  
  importFrom("stats", "na.omit")  
to your NAMESPACE file.
```

👉 So we need to import this function of the package **{stats}**

```
#' @import stats
```

or,

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

We will use the latest but by doing that it's recommended to call the function **na.omit()** as follow:

```
stats::na.omit()
```

Package dependencies

```
moyenne <- function(x, na_rm = FALSE) {  
  # Check 'x' argument [...] ----  
  # Check 'na_rm' argument [...] ----  
  
  # Remove NA (if required) ----  
  if (any(is.na(x))) {  
    if (na_rm) {  
      x <- stats:::na.omit(x)  
      if (length(x) < 2) {  
        stop("x has < 2 non-NA values.")  
      }  
    } else {  
      stop("x contains NA values. Use na_rm = TRUE.")  
    }  
  }  
  
  # Compute mean ----  
  sum(x) / length(x)  
}
```

👉 Note that the functions `any()`, `is.na()`, `sum()`, `length()` and `stop()` are in the package `{base}`

Package dependencies

```
#' Arithmetic Mean
#'
#' This function computes the arithmetic mean of a numerical vector.
#'
#' @param x a numerical vector (can contain 'NA' values).
#' @param na_rm a logical value indicating whether 'NA' values should be
#' stripped before the computation proceeds.
#'
#' @return The arithmetic mean of the values as a numeric vector of length one.
#'
#' @details An error will be returned if 'x' contains 'NA' values and 'na_rm' is
#' set to 'FALSE' or if 'x' contains less than two values (after removing 'NA').
#'
#' @export
#' @importFrom stats na.omit
#'
#' @examples
#' \dontrun{
#' moyenne(c(1:10, NA)) # error
#' }
#' moyenne(c(1:10, NA), na_rm = TRUE)

moyenne <- function(x, na_rm = FALSE) { ... }
```

Update NAMESPACE

```
devtools::document()
```

✓ Writing NAMESPACE

Here is the content of the NAMESPACE file

```
export(moyenne)
importFrom(stats,na.omit)
```

👉 The NAMESPACE controls what happens when our package is loaded but not when it's installed. This is the role of DESCRIPTION and we need to add dependencies to this file.

Update DESCRIPTION

```
usethis::use_package("stats")
```

- ✓ Adding 'stats' to Imports field in DESCRIPTION
- Refer to functions with `stats::fun()``

Here is a snippet of the **DESCRIPTION** file

```
Package: rpkg
Title: A Minimal But Complete R Package
Version: 0.0.0.9000
(...)
Imports:
  stats
Suggests:
  testthat
```

Dependencies types

- **Imports**: packages listed in this field are installed when your package is installed but are not attached when your package is attached (they are just loaded). **ALWAYS** use this method and refer to external functions using the syntax `package::function()`
- **Depends**: packages listed in this field are installed when your package is installed and are attached when your package is attached. **NEVER** use **Depends** and always use **Imports** (except for special cases)
- **Suggests**: packages listed in this field are **not** installed when your package is installed. Your package can use these packages, but doesn't require them (e.g. to run tests, build vignettes, etc.)

👉 Resources: [R Package - Chap8.1](#) and [R Package - Chap13.6](#)

Always use the function `usethis::use_package()`

```
usethis::use_package(package, type = "Imports")                                # Default  
usethis::use_package(package, type = "Suggests")
```

Let's check

```
devtools::check()
```

```
— R CMD check results ━━━━━━ rpkg 0.0.0.9000 ━━━━━━  
Duration: 11.9s  
  
0 errors ✓ | 0 warnings ✓ | 0 notes ✓
```



Version and Release



Version number

- Our package has the version number `0.0.0.9000`: this means that it's under development. But we think our package is enough stable to be released so we want to upgrade its version number
- A version number has four components: `major.minor.patch.dev`
 - `major` is reserved for changes that are not backward compatible
 - `minor` is reserved for changes introducing new features
 - `patch` is reserved for changes fixing bugs
 - When releasing a package the `dev` number must be deleted

👉 Resources: [R Package - Chap20](#)

```
usethis::use_version(which = "minor")
```

✓ Setting Version field in DESCRIPTION to '0.1.0'

Inform users – NEWS

- 👉 Inform existing users of changes in each release of your package with the `NEWS.md`

```
usethis::use_news_md()
```

- ✓ Writing '`NEWS.md`'
- Modify '`NEWS.md`'

- For instance

```
# rpkg 0.1.0
- Add new feature: `rpkg::moyenne()` computing the arithmetic mean of a
numerical vector.
```

- 👉 Try to keep this file updated after each version incrementation

```
usethis::use_git(message = ":package: Release v0.1.0")
```

Share on GitHub



Connecting to GitHub

Here we opted for creating a local versioned  project: this workflow is called **GitHub last**.

👉 So we need to:

1. Create an new empty repository on our GitHub account (or organization)
2. Add the remote (GitHub repository URL) to our local  project
3. Push our versioned  project to this new repository

There is two way of doing that:

- On the GitHub website, then on a terminal
- Directly from  in one command line

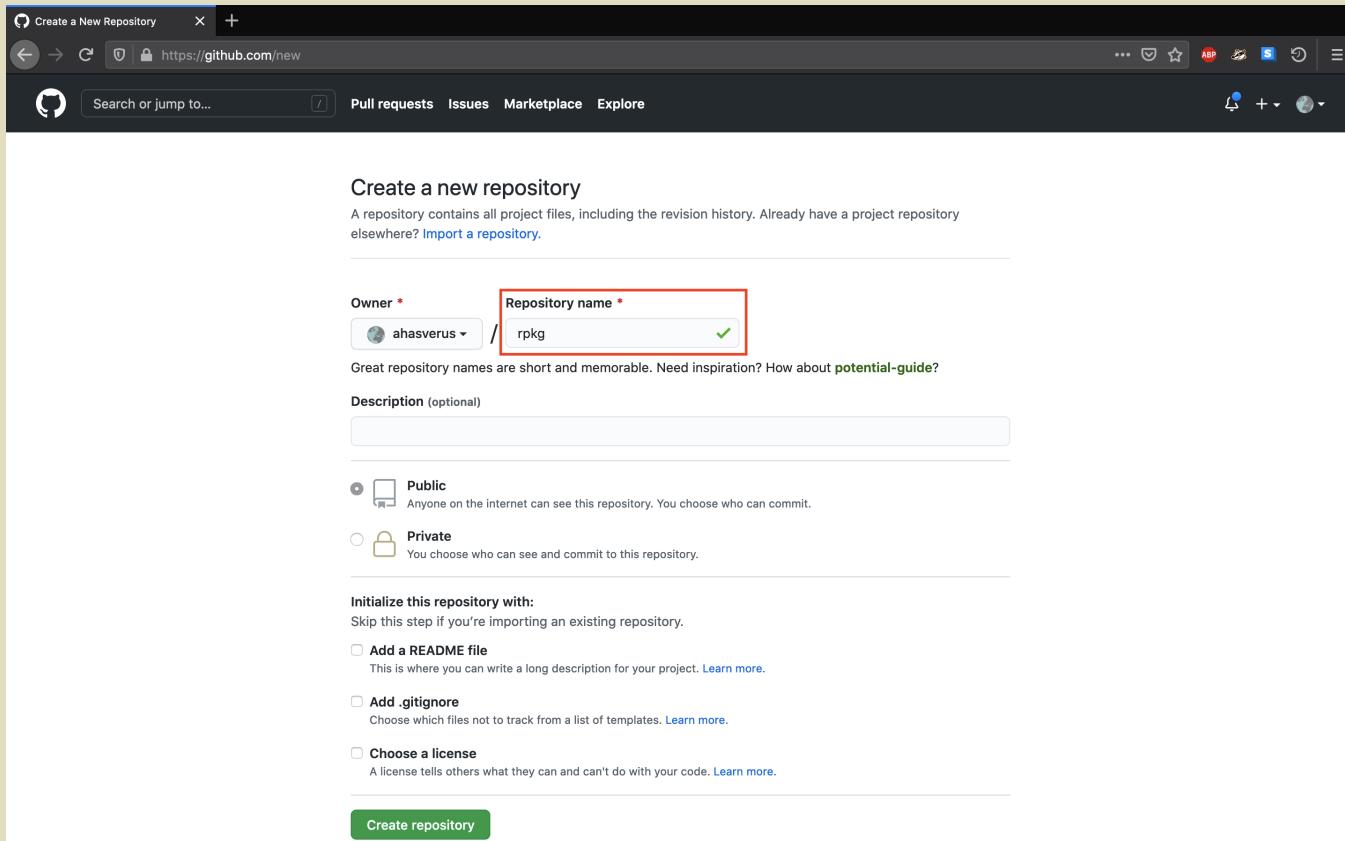
Connecting to GitHub - Option 1

👉 Log in on your GitHub account and create a new empty repository (click on the + on the top-right)

The screenshot shows a GitHub user profile for 'ahasverus (Nicolas Casajus)'. The profile includes a large profile picture of a globe, the user's name 'Nicolas Casajus', their GitHub handle 'ahasverus', and their title 'Data scientist'. Below this is an 'Edit profile' button. The stats show '27 followers', '34 following', and '15' contributions. The bio section includes 'FRB-Cesab', 'Montpellier, France', an email address, and a website URL. The 'Highlights' section mentions 'Arctic Code Vault Contributor'. On the right, there's a navigation bar with 'Overview', 'Repositories 33', 'Projects', and 'Packages'. A red box highlights the top-right corner where a '+' icon is located, which is part of a dropdown menu. The menu items are 'New repository' (highlighted in blue), 'Import repository', 'New gist', 'New organization', and 'New project'.

Connecting to GitHub - Option 1

- 👉 Give a name to the new repository and let other options to their defaults values (no README, no LICENSE, no .gitignore)



The screenshot shows the GitHub 'Create a New Repository' interface. The 'Repository name' field is highlighted with a red border and contains the value 'rpkg'. The 'Owner' dropdown is set to 'ahasverus'. Below the form, there's a note about repository names being short and memorable, followed by a 'Description (optional)' input field and a choice between 'Public' and 'Private' repository types. Under 'Initialize this repository with:', there are four optional checkboxes: 'Add a README file', 'Add .gitignore', and 'Choose a license', each with its own explanatory text. At the bottom is a green 'Create repository' button.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Owner * / Repository name *

Great repository names are short and memorable. Need inspiration? How about [potential-guide](#)?

Description (optional)

 Public Anyone on the internet can see this repository. You choose who can commit.

 Private You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.

Add a README file This is where you can write a long description for your project. [Learn more](#).

Add .gitignore Choose which files not to track from a list of templates. [Learn more](#).

Choose a license A license tells others what they can and can't do with your code. [Learn more](#).

Connecting to GitHub - Option 1

- Choose your protocol with which you communicate with GitHub (HTTPS or SSH if you have generated SSH keys)

The screenshot shows a GitHub repository page for 'ahasverus/rpkg'. At the top, there's a navigation bar with links for Pull requests, Issues, Marketplace, and Explore. Below that is a header bar with the repository name 'ahasverus / rpkg' and metrics for Unwatch (1), Stars (0), Forks (0). The main content area has a heading 'Quick setup — if you've done this kind of thing before' followed by a section for 'Set up in Desktop' with options for HTTPS and SSH. A red circle highlights the 'SSH' button. Below this, there's a command-line guide for creating a new repository:

```
echo "# rpkg" >> README.md  
git init  
git add README.md  
git commit -m "first commit"  
git branch -M main  
git remote add origin git@github.com:ahasverus/rpkg.git  
git push -u origin main
```

Further down, another section is highlighted with a red box, titled '...or push an existing repository from the command line', containing the following command:

```
git remote add origin git@github.com:ahasverus/rpkg.git  
git branch -M main  
git push -u origin main
```

At the bottom, there's a section for importing code from another repository.

Connecting to GitHub - Option 1

- 👉 To connect a local git project to a GitHub repository, run the following command line under the RStudio terminal*:

```
git remote add origin git@github.com:ahasverus/rpkg.git
```

or, if you prefer directly on the  console:

```
system("git remote add origin git@github.com:ahasverus/rpkg.git")
```

Since October 2020, GitHub has decided to replace the 'master' branch with 'main' in order to remove unnecessary references to slavery and replace them with more inclusive terms (see [here](#) and [here](#)).

- 👉 So we need to rename our local 'master' branch in 'main'

```
git branch -M main
```

* **ahasverus** is my GitHub account and **rpkg** my package/repository name

Connecting to GitHub - Option 1

👉 Before pushing our local project we will add two lines in the **DESCRIPTION** file:

URL: <https://github.com/ahasverus/rpkg>
BugReports: <https://github.com/ahasverus/rpkg/issues>

Let's commit our changes and push our project on GitHub

```
usethis::use_git(message = ":bulb: Update package metadata")
system("git push -u origin main")
```



Connecting to GitHub - Option 2

An alternative to previous steps is the function `usethis::use_github()`. This function will:

1. Create a new repository
2. Add the remote locally
3. Edit the **DESCRIPTION**

All in one command line. Pretty cool, isn't it!

👉 **BUT** first we need to get a **GitHub Personal Access Token** (PAT) which will allow  to control GitHub (only once)

```
usethis::browse_github_token()
```

Generate your token and copy its value

Connecting to GitHub - Option 2

Once our new GitHub PAT is generated, it is common to store it as an environment variable in the file `.Renvironment`

- Let's open this `.Renvironment` file

```
usethis::edit_r_environ()
```

- And add the following line:

```
GITHUB_PAT=999999zzz99zz9z9zzzz9zzz99999z9z9z9zz99
```

👉 Make sure to replace `999999zzz...` by your token and make sure `.Renvironment` ends with a newline!

Verification (after restarting RStudio):

```
usethis::github_token()
```

```
## [1] "999999zzz99zz9z9zzzz9zzz99999z9z9z9zz99"
```

👉 Resources: **Managing secrets**

Connecting to GitHub - Option 2

We are ready to connect our local project to new Github repository

```
usethis::use_github(protocol = "ssh")
usethis::use_github(protocol = "https")      # If you haven't generated SSH keys
```

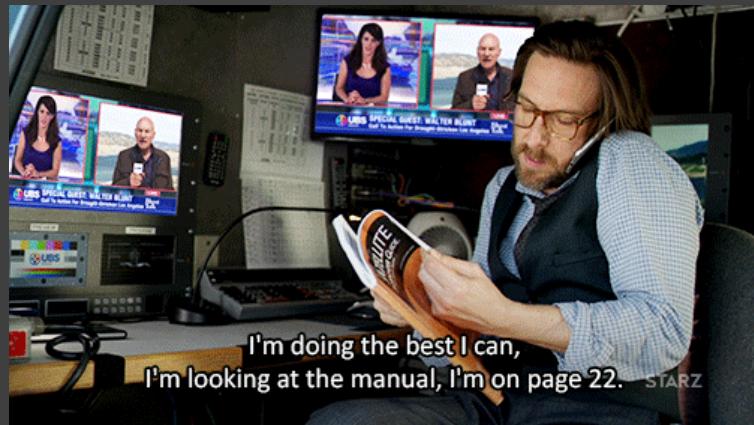
- ✓ Checking that current branch is 'master'
- ✓ Creating GitHub repository
- ✓ Setting remote 'origin' to 'git@github.com:ahasverus/rpkg.git'
- ✓ Adding GitHub links to DESCRIPTION
- ✓ Setting URL field in DESCRIPTION to 'https://github.com/ahasverus/rpkg'
- ✓ Setting BugReports field in DESCRIPTION to 'https://github.com/ahasverus/rpkg/issues'
- ✓ Pushing 'master' branch to GitHub and setting remote tracking branch
- Failed to push and set tracking branch.

👉 This function will also try to push project but it will fail. We need to run:

```
system("git push --set-upstream origin master")
```

Now our local project and Github are linked and synchronized (for now). After committing future changes we will be able to push normally (using RStudio git panel or `git push`)

Add a README



Add a README

👉 It's time to add a **README** to our repository: in a glance user will know more about our project (what it does, how to install it, and how to use it).

```
usethis::use_readme_rmd()
```

- ✓ Writing 'README.Rmd'
- ✓ Adding '^README\\\\.Rmd\$' to '.Rbuildignore'
- Modify 'README.Rmd'

You can find an example of the `README.Rmd` of our package [here](#)

We are creating a `.Rmd` because our project is an  package. After editing the file we'll need to compile it into a `.md` file.

```
rmarkdown::render("README.Rmd")
```

- ✓ Preview created: `README.html`
- ✓ Output created: `README.md`

Add a README

Two files have been created. The `.md` will be used by GitHub and the CRAN, but the `.html` will not be used by  during the build of our package.

Let's add it to the `.Rbuildignore` (and the `.gitignore`)

```
usethis::use_build_ignore("README.html")
usethis::use_git_ignore("README.html")
```

```
usethis::use_git(message = ":pencil: Edit README")
system("git push")
```

 Each time you edit the `README.Rmd` you will have to update the `.md` with `rmarkdown::render("README.Rmd")`

The Wonderful world of the GitHub Actions

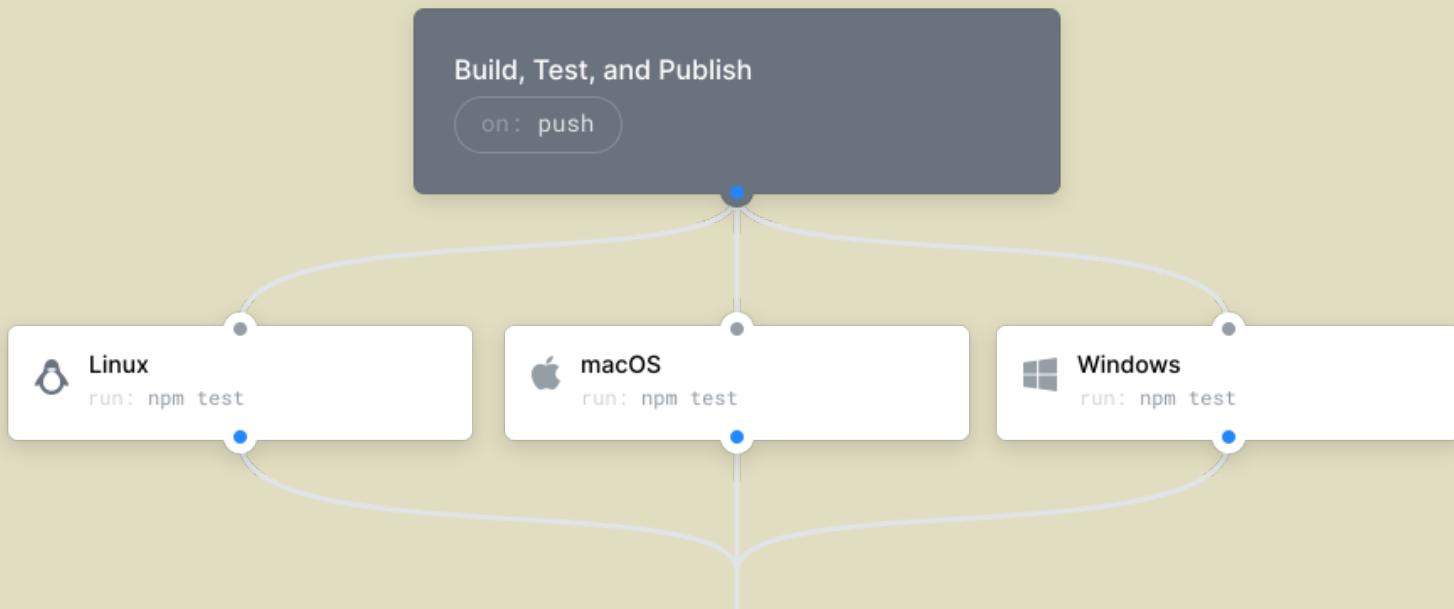


SENORGIF.COM

GitHub Actions

And if we were able to trigger automated tasks after we push to GitHub?

👉 This is where  GitHub Actions come in. They can trigger actions that will automate our workflow under a CI/CD framework allowing us to build (check), test and deploy our  package after each interaction with GitHub.



👉 Resources: [Official website](#)

GH Actions - Checks & Tests

👉 Instead of locally (and each time we make changes) checking our  package with `devtools::check()` and running unit tests with `devtools::test()`, we can delegate these tasks to GitHub with the functions of the family: `usethis::use_github_action_check_*`()

You can replace `*` by one of:

- `release`: this will build our package on macOS with the latest release of 
- `standard`: this will build our package on three OS (linux, macOS and Windows) and the release and devel version of 
- `full`: this will build our package on the same three OS and multiple versions of 

GH Actions - Checks & Tests

👉 Let's check and test our package in a robust way (cross-platform and multiple versions of 

```
usethis::use_github_action_check_full()
```

- ✓ Creating '.github/'
- ✓ Adding '*.html' to '.github/.gitignore'
- ✓ Creating '.github/workflows/'
- ✓ Writing '.github/workflows/R-CMD-check.yaml'
- ✓ Adding R build status badge to 'README.Rmd'
- Re-knit 'README.Rmd'

The configuration file for this GH Actions has been stored in **.github/workflows/** (a **YAML** file): it will trigger the check and test our package once we have committed and pushed changes.

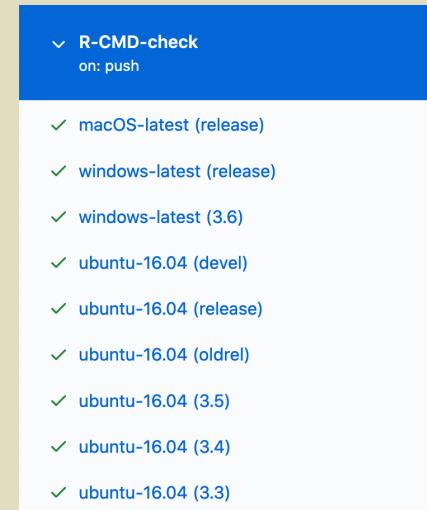
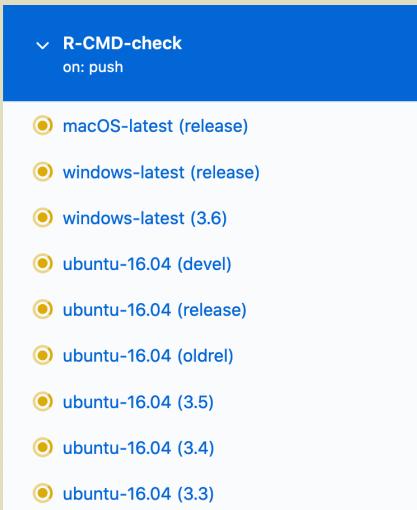
👉 Our **README.Rmd** has been modified: a badge has been added. So we need to re-knit it:

```
rmarkdown::render("README.Rmd")
```

GH Actions - Checks & Tests

👉 Let's commit changes and push to our repository

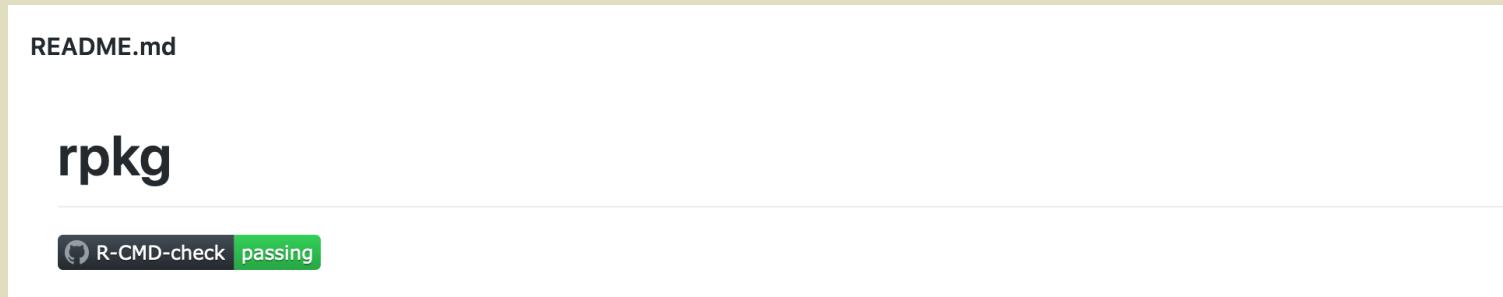
```
usethis::use_git(message = ":octocat: Configure gh-actions (checks & tests)")  
system("git push")
```



👉 After a few minutes our package has been checked and tested by GitHub servers!

GH Actions - Checks & Tests

👉 And we've got a green badge. **Yeah!!!**



👉 Resources: **GH Actions for R**

👉 And the complete `_devhistory.R` for this package: **here**

To go further...



To go further...

- 👉 A website for your package [Lien](#)
- 👉 Add badges to your repository [Lien](#)
- 👉 Add a DOI with Zenodo [Lien](#)
- 👉 Create a logo for you packages [Lien](#)
- 👉 Add Data to your package [Lien](#)
- 👉 Add a vignette (tutorial) [Lien](#)
- 👉 Add a code coverage [Lien](#)

That's all folks!

