R Configuration Models

Last Update: 12-Aug-25

Contents

Definitions

Project has two meanings we need to clarify. To avoid confusion, we will use the word "study" when referring to the second meaning, even if the work you are doing is not a 'study.'

- R Project (.Rproj extension) – An R file that allows relative referencing of files in a directory and subdirectory without hard coded "setwd" commands and provides other benefits. https://www.r-bloggers.com/2020/01/rstudio-projects-and-working-directories-a-beginners-guide/

- Project meaning a set of activities focused on a goal (e.g. The project to study stamp collecting as a neurosis.) https://www.merriam-webster.com/dictionary/project

Summary

This paper covers a simple model for creating a multi-layer configuration for R users. It applies to single and multi-user environments. The model handles both 'global' configurations, as well as project-specific configurations. The paper assumes a working knowledge of R syntax.

Background

If you are doing work that requires consistency, and you want to reduce code duplication then having a method to centralize settings and values across users and programs then having a configuration process is critical. We will cover the general design of the model presented, then look deeper into specific uses. My model was created from work I did at NEHRI where we had dispersed users and dispersed working environments. We needed a way to have code know where certain files were stored depending on the environment. We also wanted to use template programs that could rely on functions working consistently without user intervention.

Design Concept

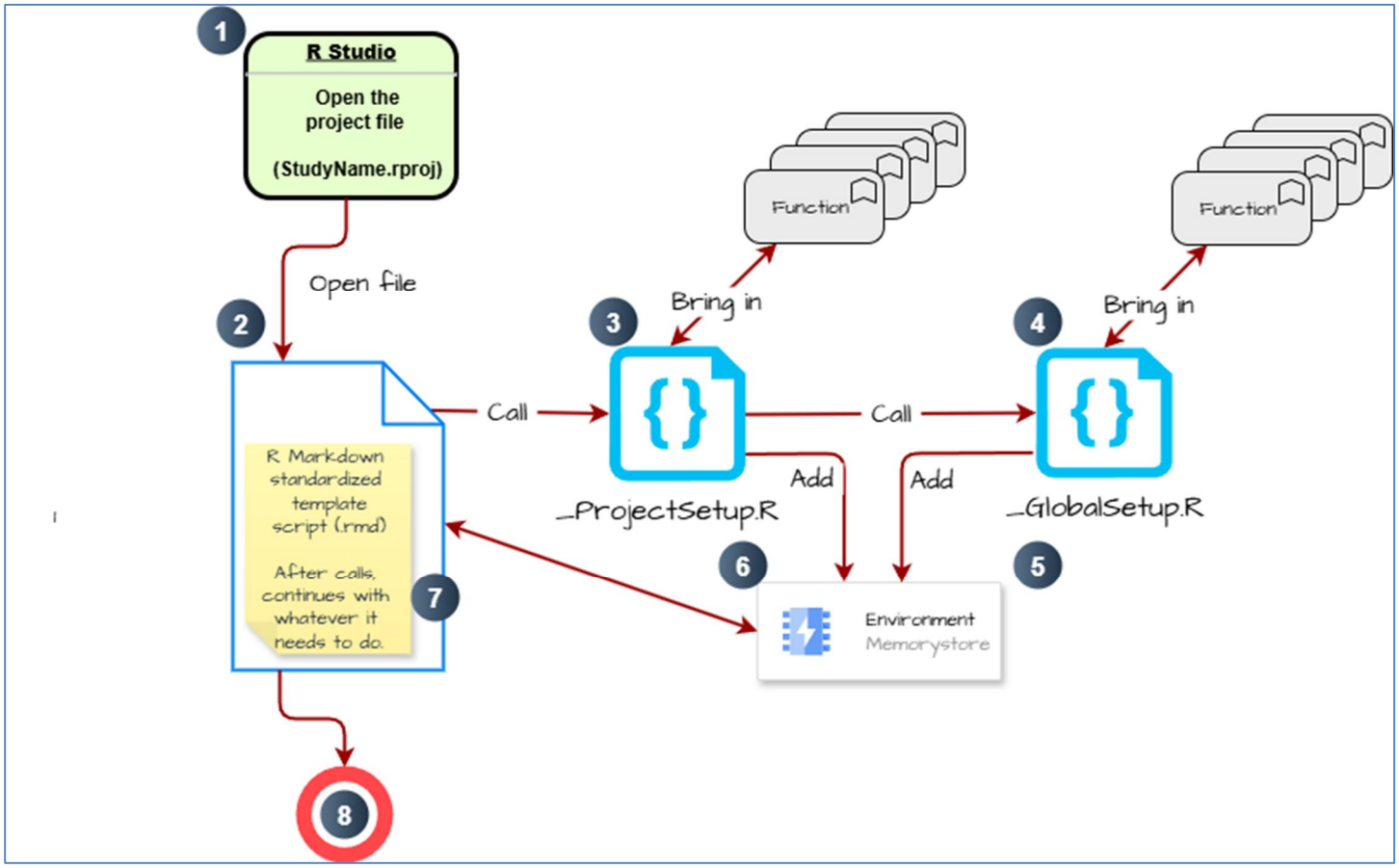
We start at the bottom and move up. The first step is having a "study" level configuration file. This is the lowest level and does require some code that provides the system with information that it cannot know about on its own. An example is a folder that is NOT a subdirectory of that area.

The beauty of the project concept in R is that the working directory is set based on the project folder and even if the folder moves or is duplicated across environments (such as Dropbox local copies), the relative references of subfolders do not change. But if you have an object (such as a set of huge reference data files) you do not want to copy them into every project that uses them just so the files can be referenced as a subdirectory of each project.

The next step up the chain is a global configuration file. This can contain values that every project should have access to, or every user's R session should have as an initial setting. We will show examples of this for function options, and global functions not in a package, and values.

General Flow

We will now show the general flow of the parts.

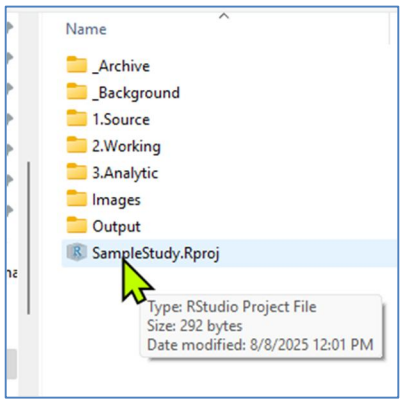


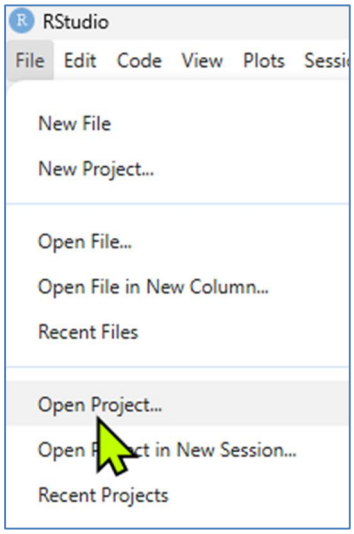
1. Start R Studio directly from the project file, or through R Studio dialog.
2. Open the R Markdown standardized template.
3. Call the study-specific "_ProjectSetup.R" file which checks where it is and then…
4. Call the "_GlobalSetup.R" file.
5. "_GlobalSetup.R" sets values, options and if there are global functions not in a package, brings them in.
6. Return to the "_ProjectSetup.R" and sets any study-specific values, study-specific options and if there are study-specific functions not in a package, brings them in.
7. The R Markdown file continues with whatever work it is meant to do.
8. End

Detail

We will now show the general flow of the parts.

1. Opening the .Rproj directly or via R Studio dialog





2. We open an R Markdown file using the R Studio dialog (not by opening from File Explorer which by passes Step 1).

3. We source (call) the "_ProjectSetup.R" file. The file resides in the same folder as the R project file.

```
source("_ProjectSetup.R")
```

"_ProjectSetup.R" Code	Explanation

`# ----------------------------------------------------------#`
`#  _ProjectSetup.R - Project specific settings and functions`
`# ----------------------------------------------------------#`
 	
`# Determine the global setup file location`
`if (nchar(Sys.getenv("Name_of_environ_Variable")) > 0) {`
`  # There is an environment variable available`
`  publicDir  <- paste0(Sys.getenv("Name_of_environ_Variable "))`
 	
`} else if (file.exists("//server1/folder1")) {`
`  # We are on the X network`
`  publicDir  <- "//server1/folder1"`
 	
`} else if (file.exists("//server2/folder1")) {`
`  # We are on the Y network`
`  publicDir  <- "//server2/folder1"`
`}`
 	
`  # You need to manually specify where the Public folder`
`  # is on your computer if not in a known environment.`
`  if (!exists("publicDir")) {`
`  publicDir <- NA`
`  }`
	If users are working on local laptops using a syncing service such as Dropbox, then the easiest is to create a Windows system environment variable that points to the Dropbox "public" location on their specific computer.
	
	Otherwise, it checks to see if they are on any number of different networks. The key point is that in each computing environment, the global folder is identified and known.
	
	The downside of this approach is that this code must be in each _projectSetup.R Without configuring each workstation and adding scripts that run at startup, R cannot know where to look without this code.

"_ProjectSetup.R" Code	Explanation

`if (exists("publicDir")&!is.na(publicDir)) {`
`  source(paste0(publicDir, "/folder/_GlobalSetup.R"), echo=FALSE)`
 	
`} else {`
`  publicDir <- ""`
`  print("*** ERROR: publicDir not found ***")`
`  print("Check if network public folder is unavailable.")`
`}`
	Now that we know the location, we can call the "_GlobalSetup.R."
	
	For now, we will explain the rest of the code in "_ProjectSetup.R" and explain "_GlobalSetup.R" in the next table.

`# ------------------------------------------------------------------------ #`
`#  EDIT AFTER THIS POINT ONLY  #`
`# ------------------------------------------------------------------------ #`
 	
`sourceDir  <- paste0(baseDir,"1.Source/")`
`workingDir  <- paste0(baseDir,"2.Working/")`
`analyticDir  <- paste0(baseDir,"3.Analytic/")`
 	
`# ---------------------------------`
`# Project specific parameters  #`
`# (those used in multiple programs) #`
`# Edit below as needed, remove vars #`
`# not needed in the project  #`
`# ---------------------------------- #`
 	
`# Flextable project defaults`
`flextable::set_flextable_defaults(`
`  font.size = 8, font.family = "Calibri",`
`  font.color = "#333333",`
`  table.layout = "autofit",`
`  border.color = "gray",`
`  padding.top = 3, padding.bottom = 3,`
`  padding.left = 4, padding.right = 4)`
 	
`  cat("Project specific parameters","\n")`
`  cat("----------------------------","\n")`
 	
`  ProjectTitle <- "Stamp Collectors Study"`
 	
`  # show in the log`
`  cat("ProjectTitle: ", ProjectTitle,"\n")`
 	
`# --------- Functions ---------- #`
 	
`# We load any files found in the project function location`
 	
`rFiles <- dir(paste0("/StudyFunctions/"), pattern="*.R")`
`for (rFile in rFiles)  {`
`  source(paste0("/StudyFunctions/", rFile))`
`}`
`rm(rFile, rFiles)`
 	
`# ------------------------------------------------------------#`
`#  _ProjectSetup.R end`
`# ------------------------------------------------------------#`
	These are examples and would be customized for your study.
	
	Here the study wants to ensure specific font and style when using FlexTable, so having this here means the entire study has consistent look to FlexTable output.
	
	We ensure the activity appears in the output by displaying any values we set.
	
	Here the study has study-specific functions (not in a package) that it wants to bring in for every program.
	
	Because this is wildcard based, you just drop files in the folder and do not have to hard code bringing them in individually.

4. The "_GlobalSetup.R" will execute before the rest of the "_ProjectSetup.R" code as explained above.

"_GlobalSetup.R" Code	Explanation

`# ----------------------------------------------------------#`
`#  _GlobalSetup.R - General global settings and functions`
`# ----------------------------------------------------------#`
 	
`# History`
`# YYYY MM DD - Author - What changed.`
 	
`# --------- Cleanup ----------- #`
 	
`# Remove any files in C:\X if it exists`
`if (base::file.exists("C:/X")) {`
`a <- base::file.remove(dir("C:/X", pattern = "\\.png$", full.names = TRUE))`
`   rm(a)`
`}`
	It is always a good idea to have a change log in a file with this importance.
	
	Here we show an example of cleaning up junk files.

`# --------- Settings ---------- #`
 	
`startingTime <- Sys.time()`
`scriptInfo   <- rstudioapi::getSourceEditorContext()$path`
`baseDir      <- stringr::str_remove(scriptInfo, basename(scriptInfo))`
 	
`# Summarytools defaults`
`summarytools::st_options(`
`  bootstrap.css     = FALSE,`
`  plain.ascii       = FALSE,`
`  style             = "rmarkdown",`
`  dfSummary.silent  = TRUE,`
`  footnote          = NA,`
`  descr.silent      = TRUE,`
`  tmp.img.dir       = "C:/X",`
`  subtitle.emphasis = FALSE)`
 	
`# 24 color pallet for plots`
`c24 <- c(`
`  "dodgerblue2", "#E31A1C", # red`
`  "green4",`
`  "#6A3D9A", # purple`
`  "#FF7F00", # orange`
`  "gold1",`
`  "skyblue2", "#FB9A99", # lt pink`
`  "palegreen2",`
`  "#CAB2D6", # lt purple`
`  "#FDBF6F", # lt orange`
`  "gray70", "khaki2",`
`  "maroon", "orchid1", "deeppink1", "blue1", "steelblue4",`
`  "darkturquoise", "green1", "yellow4", "yellow3",`
`  "darkorange4", "brown"`
`)`
	Here we can set global options and values.
	
	We tell R what name to use if we need to refer to the calling R Markdown program. We set a starting time in case we want to time how long the session tool (as part of knitting).
	
	We set up defaults for anyone using SummaryTools to ensure consistent output and functionality.
	
	We create a 24-color pallet, so the users do not have it. Again, this is just an example.
	
	It could be a variable with the copyright notice to include in every output, etc.

`# --------- Libraries ---------- #`
 	
`if (grepl("//server1/", publicDir)) {`
`  # Environment 1`
`  .libPaths(c(.libPaths(), "//server1/folder/R_Packages"))`
`} else if (grepl("//server2/", publicDir)) {`
`  # Environment 2`
`  .libPaths(c(.libPaths(), "//server2/folder/R_Packages"))`
`}`
	If the environment does not allow the user to install packages but does allow packages to be copied to the network (I know, that sounds crazy), this approach can add a path to other packages locations.

"_GlobalSetup.R" Code	Explanation

`# --------- Functions ---------- #`
 	
`# We load any files found in the study function folder location`
 	
`rFiles <- dir(paste0("/StudyFunctions/"), pattern="*.R")`
`for (rFile in rFiles)  {`
`  source(paste0("/StudyFunctions/", rFile))`
`}`
`rm(rFile, rFiles)`
 	
`# ------------------------------------------------------#`
`#  _GlobalSetup.R end`
`# ------------------------------------------------------#`
	Here the study has study-specific functions (not in a package) that it wants to bring in for every program.

5. and 6. As noted above, the "_GlobalSetup.R" modifies the environment, and then the remaining "_ProjectSetup.R" code modifies the environment.

6. The calling R Markdown program now does whatever it is meant to do. The key point is that whatever program code is written, should be after calling "_ProjectSetup.R" so the global- and study-specific information are already available.

7. The End!

Thoughts for the future

This method is not without drawbacks. Some of the things coded here would be unnecessary if computing environments were automatically standardized and had what users needed. This method can help get around those limitations. I am sure better minds will look and chuckle at how kludgy this is. Drop me a note, I would be glad to incorporate any better practices/better code methods.