{golem} has a function called run\_app(), which, in its original implementation by [Vincent](https://github.com/ThinkR-open/golem/commit/33ca7df69b6c8b622ba409d5435960c56c1f3ced), relies on calling runApp() on the two files contained in the inst/ folder of the package. But here comes the issue — **runApp() can’t be used with RStudio products (Shiny Server, Connect & Shinyapps.io)**, as it returns an error Can't call runApp() from within runApp(). The previous workaround was to use shinyApp(...) in the app.R file, but I like things to be stable so having various solutions didn’t seem like the best answer.

Two questions arise from that:

Why on earth a run\_app() function? Well, because it allows to deploy easily through command line. Using a run\_app() function allow more flexibility, if we succeed to design it to take parameters and to pass it to the app. For example, in Docker, we could use environment variables as function parameters. A flexible run\_app() function would also allow to change the behaviour of the app without having to tweak the app.R file we’re using for RStudio Products. Or, as you will see in this very blog post, a run\_app() function with parameters can also be used to change the behaviour of our local app 

Ok, so that was the story behind run\_app(). But why on earth this first implementation with runApp()? You’ll see in a minute why one would choose that function But, TL;DR, runApp is able to use local options defined into the function.

And these three functions (the one from the title) do not have the same behaviour, depending on where they are used:

* runApp() doesn’t work on RStudio products, but is the only way Docker and local calls can access options passed to the run\_app() function.
* shinyApp() and shinyAppDir() works likewise wherever you use them.
* RStudio production do not handle “local” options defined in the run\_app() function.
* There’s a fourth way, the one implemented in {golem}, which is fit for all scenario 

**Several ways to launch a Shiny App**

1. runApp() (old implementation of {golem}), which is a wrapper aroundshiny::runApp(system.file("app", package = "aaaaaa")).
2. shinyApp(), which is shiny::shinyApp(ui = app\_ui(), server = app\_server), the solution created by golem::add\_rstudioconnect\_file() and friends.
3. shinyAppDir(), which is shinyAppDir( system.file("app", package = "aaaaaa") ) — a necessary workaround for Shiny server if you wanted to call the app/ folder through the old implementation of {golem}.

Note: there is also shinyAppFile(), but its inner behaviour is the same as shinyAppDir(), so it’s not benchmarked here.

**One naive implementation**

**What we could have done there is simply leaving things open for the end user**, so that they have to chose the best implementation for their deployment use case. By doing something like:

run\_app <- function(

with = c("shinyApp", "runApp", "shinyAppDir")

) {

with <- match.arg(with)

if (with == "shinyApp"){

shiny::shinyApp(ui = app\_ui(), server = app\_server)

} else if (with == "runApp") {

shiny::runApp(system.file("app", package = "aaaa"))

} else if (with == "shinyAppDir") {

shiny::shinyAppDir(system.file("app", package = "aaaa"))

}

}

This might be the best answer as it leaves the choice to the user, but the question is still open: **what function should I use for my deployment**?

Let’s keep this function and use it for our benchmark.

**Side node**

If I refer to ?shinyApp:

You generally shouldn’t need to use these functions to create/run applications; they are intended for interoperability purposes.

[*https://shiny.rstudio.com/reference/shiny/1.3.2/shinyApp.html*](https://shiny.rstudio.com/reference/shiny/1.3.2/shinyApp.html)

So according to the documentation we should rarely call shinyApp() directly, and use only runApp() instead. But using runApp() is impossible on RStudio platforms, as they print an error that looks like this:

Loading aaaa

Error in shiny::runApp(system.file("app", package = "aaaa")) :

Can't call `runApp()` from within `runApp()`. If your application code contains `runApp()`, please remove it.

Calls: runApp ... eval -> eval -> ..stacktraceon.. -> run\_app ->

**Side note 2**

I’ve tried to make this benchmark as reproducible as possible, so **feel free to run it and see if you get the same results** 

The package is named “aaaa” (so it won’t conflict with any other package (I hope), and can be found [here](https://github.com/ColinFay/golem4bench). It contains the golem skeleton with the functions listed below.

**Looking for the best implementation**

Anyway, let’s try to find the best implementation. The idea is that our implementation of run\_app() should:

* **work on the maximum number of services** (Locally + Docker + RStudio products), and this with minimal tweaking (one implementation to rule them all would be best).
* **Be able to read options from the global environment**, so that for example we can use the global golem.app.prod variable from inside the server and UI, or global options / env var defined in the service.
* Be able to read **options from the local function environment**, so we can pass arguments to the run\_app() function.

**Benchmark conditions**

* Previous version of {golem} (0.0.1.600)

**Content of the app\_ui function:**

app\_ui <- function() {

tagList(

fluidPage(

h1("aaaa"),

h3( "global options:" ),

verbatimTextOutput("global"),

h3( "function options:" ),

verbatimTextOutput("shinycall")

)

)

}

**Content of the app\_server function:**

app\_server <- function(input, output,session) {

output$global <- renderPrint({

# Global options

getOption('golem.pkg.name')

})

output$shinycall <- renderPrint({

# Local options

getOption('shinycall')

})

}

**Various run\_app implementations**

We’ll use this function to benchmark the three functions (shinyApp(), runApp(), and shinyAppDir().

run\_app <- function(

with = c("shinyApp", "runApp", "shinyAppDir")

) {

with <- match.arg(with)

# Here, we set local options so we can pass

# arguments to the run\_app() function

options("shinycall" = with)

on.exit(

options("shinycall" = NULL)

)

if (with == "shinyApp"){

shiny::shinyApp(ui = app\_ui(), server = app\_server)

} else if (with == "runApp") {

shiny::runApp(system.file("app", package = "aaaa"))

} else {

shiny::shinyAppDir(system.file("app", package = "aaaa"))

}

}

**In our results, runApp is:**

run\_app( "runApp" )

**shinyApp is:**

run\_app( "shinyApp" )

**shinyAppDir is**

run\_app( "shinyAppDir" )

**Launch contexts**

**Local launch**

You can run this in your console, here in RStudio.

# Set options here

options( "golem.pkg.name" = "aaa")

# Detach all loaded packages and clean your environment

golem::detach\_all\_attached()

# rm(list=ls(all.names = TRUE))

# Document and reload your package

golem::document\_and\_reload()

# Run the application

aaaa::run\_app(with = "runApp")

**The Dockerfile for local test is**

FROM rocker/tidyverse:3.6.0

RUN R -e 'install.packages("remotes")'

RUN R -e 'remotes::install\_github("r-lib/remotes", ref = "97bbf81")'

RUN R -e 'remotes::install\_cran("shiny")'

COPY aaaa\_\*.tar.gz /app.tar.gz

RUN R -e 'remotes::install\_local("/app.tar.gz")'

EXPOSE 80

CMD R -e "options('shiny.port'=1234,shiny.host='0.0.0.0', 'golem.pkg.name' = 'aaa');aaaa::run\_app( 'runApp' )" # also with shinyApp & shinyAppDir

You can find this Dockerfile in the inst/dockerfilelocal folder of the golem4bench repo. Before launching it, **you have to document, and build your package** with devtools::build(path = "inst/dockerfilelocal/"),

The full thing can be launched with:

R -e "devtools::build(path = 'inst/dockerfilelocal/')" \

&& cd inst/dockerfilelocal/ \

&& docker build -t aaa . \

&& docker run --name aaaa -p 1234:1234 -d aaa \

&& sleep 2 \

&& open http://0.0.0.0:1234

Then, stay in the folder, change the "runApp" arg in the Dockerfile to "shinyApp", **rebuild and relaunch** from the docker build line. Then again with "shinyAppDir". Of course, don’t forget to docker kill aaa && docker rm aaa between each iteration.

**RStudio products 1/2: the app.R script**

pkgload::load\_all()

options( "golem.pkg.name" = "aaa" )

run\_app( "runApp" ) # also with shinyApp & shinyAppDir

Each three versions (i.e the three versions of run\_app()) of this file will be deployed to:

* **local Shiny Server** (copied inside the Docker)
* **ThinkR internal RStudio Connect** (sent with rsconnect::deployApp())
* **ThinkR’s shinyapps.io account** (sent with rsconnect::deployApp())

**This file should be put at the root of your package.**

**RStudio products 2/2: Setting a Shiny server for testing**

This Dockerfile can be found in the inst/dockerfileshinyserver folder of the package.

FROM rocker/shiny:3.6.0

RUN R -e 'install.packages("remotes")'

RUN R -e 'remotes::install\_github("r-lib/remotes", ref = "97bbf81")'

RUN R -e 'remotes::install\_cran("shiny")'

RUN apt-get update && apt-get install libssl-dev libxml2-dev -y

RUN R -e 'remotes::install\_cran("attachment")'

RUN R -e 'remotes::install\_github("thinkr-open/golem")'

COPY . /srv/shiny-server/aaaa

RUN cd /srv/shiny-server/aaaa && R -e "attachment::install\_from\_description()"

From the root of the package:

mv inst/dockerfileshinyserver/Dockerfile Dockerfile \

&& docker build -t plop . \

&& docker run --name plop -p 3838:3838 -d plop \

&& sleep 2 \

&& open http://0.0.0.0:3838/aaaa

Then, change for "shinyApp" and "shinyAppDir" in the app.R file, then rerun the docker build. Don’t forget to kill & rm files between each iteration, and to mv back the Dockerfile where it belongs.

**Results**

Global options are the one defined outside of run\_app(), local options are the one defined inside the run\_app().

: the app launches   
: the app doesn’t launch  
: the global options are read   
: the global options are not read   
: the function options are read  
: the function options are not read

| **Where** | **runApp** | **shinyApp** | **shinyAppDir** |
| --- | --- | --- | --- |
| Locally | 🚀📗📒 | 🚀📗⛔ | 🚀📗⛔ |
| Docker | 🚀📗📒 | 🚀📗⛔ | 🚀📗⛔ |
| Connect | 💥❌⛔ | 🚀📗⛔ | 🚀📗⛔ |
| shinyApps.io | 💥❌⛔ | 🚀📗⛔ | 🚀📗⛔ |
| ShinyServer | 💥❌⛔ | 🚀📗⛔ | 🚀📗⛔ |

So to sum up :

* Docker containers don’t get local options from the functions unless called with runApp(). Which you can verify with running in any terminal: R -e "options('shiny.port'=1234,shiny.host='0.0.0.0', 'golem.pkg.name' = 'aaa');aaaa::run\_app( 'runApp' )"
* runApp() fails on RStudio Product.
* RStudio products don’t get local options with any solution (we’ll see in a minute that’s because we can’t use runApp().

**Where do we go from there?**

So, why this different behaviours? Actually, it’s because of what shinyApp() and shinyAppDir() return, compared to runApp(). If we look at the source code of shinyApp(), the last line of code looks like this:

> shiny::shinyApp

function (ui = NULL, server = NULL, onStart = NULL, options = list(),

uiPattern = "/", enableBookmarking = NULL)

{

[...]

structure(list(httpHandler = httpHandler, serverFuncSource = serverFuncSource,

onStart = onStart, options = options, appOptions = appOptions),

class = "shiny.appobj")

}

We can see that the last thing returned by the function is a structure of class shiny.appobj, whereas the runApp() returns a running process. So the “launch” of the app with shinyApp() is not the same the the one from runApp()— the first returns an object, while the second returns a process. So the launch of the app, with shinyApp(), is actually done with print.shiny.appobj. Which is why if you do a <- shinyAppDir(appDir = "inst/app/"), you’ll not get the app running until you try to print a. Which also explains why the local options (defined inside the function) are not read: with shinyApp(), the function does return an object, so the function has ended, and the options defined there are not accessible anymore.

Why is it a good news? Let’s have a look at shiny:::print.shiny.appobj:

> shiny:::print.shiny.appobj

function (x, ...)

{

opts <- x$options %OR% list()

opts <- opts[names(opts) %in% c("port", "launch.browser",

"host", "quiet", "display.mode", "test.mode")]

args <- c(list(quote(x)), opts)

do.call("runApp", args)

}

So here the cool thing is that we can hack the x passed to the print method to add golem.options inside it, i.e. in the appOptions of the app object. Hence:

with\_golem\_options <- function(app, golem\_opts){

app$appOptions$golem\_options <- golem\_opts

app

}

and

run\_app <- function(...) {

with\_golem\_options(

app = shinyApp(ui = app\_ui(), server = app\_server),

golem\_opts = list(...)

)

}

And with a full app:

library(shiny)

options("golem.app.name" = "aaa")

get\_golem\_options <- function(which = NULL){

if (is.null(which)){

getShinyOption("golem\_options")

} else {

getShinyOption("golem\_options")[[which]]

}

}

with\_golem\_options <- function(app, golem\_opts){

app$appOptions$golem\_options <- golem\_opts

app

}

app\_ui <- function() {

tagList(

fluidPage(

verbatimTextOutput("all"),

verbatimTextOutput("opt"),

verbatimTextOutput("glob")

)

)

}

app\_server <- function(input, output,session) {

output$all <- renderPrint({ get\_golem\_options() })

output$opt <- renderPrint({ get\_golem\_options("a") })

output$glob <- renderPrint({ getOption("golem.app.name") })

}

run\_app <- function(...) {

with\_golem\_options(

app = shinyApp(ui = app\_ui(), server = app\_server),

golem\_opts = list(...)

)

}

run\_app(a = "pouet", b = "bing")

And the good news is… it works everywhere 

So thanks to this little hack, you are now able to use the run\_app() function from {golem} everywhere. And now, when you build your package, you can use arguments with the run\_app() function, and use them with get\_golem\_options(). Global options are, as usual, available with getOptions().

The other little cool hack? When you generate an app.R file for RStudio products with golem::add\_rstudioconnect\_file, golem::add\_shinyappsio\_file, or golem::add\_shinyserver\_file, we left a small ShinyApp in a comment, so you can use the nice blue button to deploy in just one click (yes, apparently it seems that every time RStudio sees a ShinyApp in the text, it shows this little button )