Odd Hypothesis

The other other alternative to H0

2014/04/01

Deploying Desktop Apps with R

(Update 3: 2015-03-24) In response to a comment by Edward Kwartler, I've also added extra details for how to construct an InnoSetup script.

(Update 2: 2014-04-07) Today I encounted a few issues during a deployment that warranted a few extra notes during deployment setup and changing some of the code in `runShinyApp.R`. These are noted below.

(Update) Despite the original publish date (Apr 1), this post was **not** and April Fools joke. I've also shortened the title a bit.

As part of my job, I develop utility applications that automate workflows that apply more involved analysis algorithms. When feasible, I deploy web applications as it lowers installation requirements to simply a modern (standards compliant) web-browser, and makes pushing updates relatively transparent. Occasionally though, I need to deploy a desktop application, and when I do I rely on either Python or Matlab since both offer easy ways to make executables via freezing or compiling code.

I've long searched for a similarly easy way to make self-contained desktop applications using R, where "self-contained" means users do not have to navigate installing R and package dependencies on their system (since most live under IT lockdown).

Since the start of this year (2014), I've been jotting notes on how to do this:

1. Use R-Portable as a self-contained R engine
2. Use the `shiny` R package as the UI building framework
3. Use GoogleChromePortable in "app" mode as the UI display engine

Apparently, I wasn't the first person to have this idea. Just last week I found a blog post by ZJ Dia at Analytixware describing this very strategy. Kudos to ZJ for providing a great starting point! I tested this out and documented my additions, challenges, solutions, and thoughts.

Before going on, I first should mention that the process is **Windows specific**. I don't doubt that something similar could be achieved under OS X, but I lack a Mac to test on.

Step 1: Create a deployment skeleton

Because this technique uses portable apps, you can save yourself some time by creating a skeleton desktop deployment. This skeleton can then be copied and customized for every new desktop app you create (Step 2). This is very much like using `virtualenv` with Python, in that each deployment is its own stand alone R environment and packages.

Step 1.1: Create the skeleton folder and engine framework

Create a folder called `dist/`.

Download:

- R-Portable
- Google Chrome Portable

and install both into `dist/`.

Create a folder called `dist/shiny/`. This is where the files for your Shiny app (e.g. `ui.R` and `server.R`) will reside.

► 2009 (2)

The heirarchy of `dist/` should now be:

```
dist/
+- GoogleChromePortable/
+- R-Portable/
+- shiny/
```

Step 1.2: Install Core R Packages

Before doing anything, add to the bottom of `R-Portable/App/R-Portable/etc/Rprofile.site` :

```
.First = function(){
    .libPaths(.Library)
}
```

This will force R-Portable to only use its local library (specified in the hidden global variable `.Library`) for installing/loading packages. Without this, if you have the non-portable version of R installed on your system, R-portable will detect your user library and include it in its `.libPaths()` . Then, any packages you install will be *installed in your system or user R library, and not in R-Portable*!

(In fact R-Portable detects just about all user settings - e.g. will read `.RProfile` in your user profile if you have one, so be sure there are no potential conflicts there)

Start R-Portable and install core package dependencies. At minimum this should be the package `shiny` and all its dependencies.

```
.libPaths() # verify that only the local R-Portable library path is available
install.packages('shiny')
```

UPDATE: I had a couple of instances where a package would appear to install but would end with a warning such as:

```
Warning: unable to move temporary installation 'XYZ' to 'ZYX'
```

From what I could find, this is likely due to the Windows Search indexing service or an antivirus program. Attempting the install a second (or third) time would sometimes work. Often, I would simply resort to copying the package from the temporarily stored `.zip` file, the path to which is also printed at the end of an attempted package install.

Step 1.3: Create Application Launch Scripts

To launch your application you will need two scripts:

- `runShinyApp.R` : an R-script that loads the `shiny` package and launches your app via `runApp()`
- A shell script (either a `*.bat` or `*.vbs` file) that invokes R-portable

Step 1.3.1: Create a Shiny app R launch script (`runShinyApp.R`)

In this script you need to do the following:

- Set `.libPaths()` to only point to the local R-Portable library
- Set the default web browser to the local GoogleChromePortable in “app” mode
- Launch your Shiny app

Setting `.libPaths()` to point to the local R-Portable library *should* be handled by the modification to `Rprofile.site` made above. It’s a good idea to at least print a verification that the correct library location is being used:

```
# this message is printed on several lines (one per path) to make multiple paths
# easier to spot
message('library paths:\n', paste('... ', .libPaths(), sep='', collapse='\n'))
```

Setting the default web browser is not so straight forward. I could not get GoogleChromePortable to work with a `GoogleChromePortable.ini` file defining `AdditionalParameters` as previously described. The browser would hang with a blank page and eventually crash. So I needed another way to launch GoogleChromePortable with the `--app` commandline option to enable “app” mode.

Inspecting the code and docs for `shiny::runApp()` I found that the `launch.browser=` argument can be a function that performs special browser launching operations, whose first argument is the

application's URL (path and port).

~~In my case, I needed to use `shell()` to get Chrome to run with the `--app` option without error:~~

UPDATE: While `shell()` appeared to work on my main development workstation, it didn't on my laptop where I performed a test deployment. The app would again hang at a blank but loading Chrome window. The issue was resolved when I switched to using `system2()`, which is specific to making command line calls in Windows. In addition, I used extra caution by first translating any forward slashes (`/`) to backslashes (`\`) in the command call using `chartr()`.

One could further parameterize the function to allow for either the system or portable version of Chrome to be used:

```
# both chromes work!
chrome.sys = 'C:/Program Files (x86)/Google/Chrome/Application/chrome.exe'
chrome.portable = file.path(getwd(),
                            'GoogleChromePortable/App/Chrome-bin/chrome.exe')

launch.browser = function(appUrl, browser.path=chrome.portable) {
    browser.path = chartr('/', '\\', browser.path)
    message('Browser path: ', browser.path)

    CMD = browser.path
    ARGS = sprintf('--apps=%s', appUrl)

    system2(CMD, args=ARGS, wait=FALSE)
    NULL
}
```

So if your users already have Google Chrome installed, you can potentially make the deployed installation smaller.

Finally, launching the app is straightforward:

```
shiny::runApp('./shiny/', launch.browser=launch.browser)
```

Notice that no port is specified in the call to `runApp()`, taking advantage of Shiny's default behavior of finding a random unoccupied port to serve on. (I figured this out the hard way by originally specifying `port=8888`, on which I already had an IPython notebook running).

So in sum, your `runShinyApp.R` should have the following contents:

```
message('library paths:\n', paste('... ', .libPaths(), sep='', collapse='\n'))

chrome.portable = file.path(getwd(),
                            'GoogleChromePortable/App/Chrome-bin/chrome.exe')

launch.browser = function(appUrl, browser.path=chrome.portable) {
    browser.path = chartr('/', '\\', browser.path)
    message('Browser path: ', browser.path)

    CMD = browser.path
    ARGS = sprintf('--apps=%s', appUrl)

    system2(CMD, args=ARGS, wait=FALSE)
    NULL
}

shiny::runApp('./shiny/', launch.browser=launch.browser)
```

Step 1.3.2: Create a shell launch script
This is the script that your users will double-click on to launch the application. Herein, you need to call R-portable non-interactively (aka in BATCH mode) and have it run your `runShinyApp.R` script.

The simplest way to do this is to create a Windows Batch file called `run.bat` with the following contents:

```
@echo off
SET ROPTS=--no-save --no-environ --no-init-file --no-restore --no-Rconsole
R-Portable\App\R-Portable\bin\Rscript.exe %ROPTS% runShinyApp.R 1> ShinyApp.log 2>&1
```

Notice that `Rscript.exe` is called instead of `R.exe CMD BATCH`. In my testing, I found `Rscript` to load about an order of magnitude faster (i.e. 3s instead of 40s). Indeed, it seems that `R CMD BATCH` is somewhat of a legacy carryover and that `Rscript` should be used for commandline based scripting.

Here, I've set the following options:

```
--no-save --no-environ --no-init-file --no-restore --no-Rconsole
```

which is one flag short of `--vanilla`, to allow for `Rprofile.site` settings to be loaded. Otherwise the effect is the same; R loads without reading a user profile, or attempting to restore an existing workspace, and when the script is complete, the workspace is not saved. The last part of the command — `1> ShinyApp.log 2>&1` — specifies that all output from `stdout` (`1>`) and `stderr` (`2>`) be captured by the file `ShinyApp.log`.

It should be noted that all output generated by `print()` is sent to `stdout`. Everything else — `message()`, `warning()`, `error()` — is sent to `stderr`.

If your prefer to separate "results" output (i.e. `print()` statements) from "status" output (i.e. `message()` statements and the like), you can do so by specifying different files:

```
Rscript.exe %ROPTS% runShinyApp.R 1> ShinyAppOut.log 2> ShinyAppMsg.log
```

or leave the "results" output in the console and only capture messages/errors:

```
Rscript.exe %ROPTS% runShinyApp.R 2> ShinyAppMsg.log
```

Using a `*.bat` file will leave a system console window for users to look at. They can minimize it (or you can setup a shortcut `*.lnk` to start it minimized) to get it out of the way, but it will still be in their taskbar. If a user closes it by accident, the app will terminate in a possibly ungraceful manner.

If you don't want users to see this, you can use a `*.vbs` script which provides more options on how to display the console window. Such a script would have the following contents:

```
Rexe           = "R-Portable\App\R-Portable\bin\Rscript.exe"
Ropts          = "--no-save --no-environ --no-init-file --no-restore --no-Rconsole"
RScriptFile    = "runShinyApp.R"
Outfile        = "ShinyApp.log"
strCommand     = Rexe & " " & Ropts & " " & RScriptFile & " 1> " & Outfile & " 2>&1"

intWindowStyle = 0     ' Hide the window and activate another window.'
bWaitOnReturn  = False ' continue running script after launching R   '

' the following is a Sub call, so no parentheses around arguments'
CreateObject("Wscript.Shell").Run strCommand, intWindowStyle, bWaitOnReturn
```

So now the hierarchy in `dist/` should be:

```
dist/
+- GoogleChromePortable/
+- R-Portable/
+- shiny/
+- run.(vbs|bat)
+- runShinyApp.R
```

and completes building the deployment skeleton.

Step 2: Clone and customize

To prepare a new app for deployment, copy and rename the `dist/` folder — e.g. `copy ./dist ./TestApp`. There are a lot of files in the underlying R-portable (a few thousand in fact), so this will take a minute or so.

Step 2.1: Load required dependencies

If your app requires packages other than `shiny` and what comes with base R, this is the time to install them. As before, when building the skeleton, launch the new copy of R-portable, and install any packages that your app requires.

Copy your Shiny app files (e.g. `ui.R` , `server.R`) into `TestApp/shiny/` .

Step 2.2: Prepare for termination

Before testing your application, you need to ensure that it will stop the websocket server started by `shiny::runApp()` and the underlying R process when the browser window is closed. To do this, you need to add the following to `server.R` :

```
shinyServer(function(input, output, session){
    session$onSessionEnded(function() {
        stopApp()
    })
})
```

Here, `session` is a Reference Class object that handles data and methods to manipulate and communicate with the browser websocket client. Unfortunately, documentation for this object is not provided via the typical R help system (likely related to the shifting landscape of how best to document Reference Classes). Thankfully, the source code has a well written docstring:

```
onSessionEnded = function(callback) {
      "Registers the given callback to be invoked when the session is closed
      (i.e. the connection to the client has been severed). The return value
      is a function which unregisters the callback. If multiple callbacks are
      registered, the order in which they are invoked is not guaranteed."
      return(.closedCallbacks$register(callback))
    }
```

Thus, `onSessionEnded` is a method that stores functions to be executed when the connection between the websocket server and client is severed — i.e. when the browser is closed.

Notice that explicitly stopping the R process with `q()` is unnecessary. The only thing keeping it alive is the blocking nature of `shiny::runApp()` . Once this completes via `stopApp()` the `runShinyApp.R` script will be complete and the R process spawned by `Rscript` will naturally terminate.

Step 2.3: Test!

Your application folder should have the following hierarchy:

```
TestApp/
+- GoogleChromePortable/
+- R-Portable/
+- shiny/
   +- ui.R
   +- server.R
+- run.(vbs|bat)
+- runShinyApp.R
```

and everything should be in place.

Double-clicking on either `run.vbs` or `run.bat` (whichever you created in Step 1) should launch your app. If it doesn't, check the content in `ShinyApp.log` for any errors.

Step 3: Distribute

The simplest way to distribute your application is to zip up your application folder and instruct users to unzip and double-click on `run.vbs` (or `run.bat`). While easy on you the developer, it adds a bit of complexity to the user as they'll need to remember where they unzipped your app and what file to double click to get it to run.

To distribute a bit more professionally, I agree with Analytixware's suggestion to pack your entire application folder into an installation executable using the freely available InnoSetup tool. This allows for control of where the application is installed (e.g. preferably to a user's `LocalAppData` folder without the need for admin rights escalation), as well as the generation of familiar Start Menu, Desktop, and QuickLaunch shortcuts.

The compiler script wizard in InnoSetup creates a good starting point. That said there are a couple parameters that need to be tweaked.

First, your "executable" will be either the `*.bat` or `*.vbs` script your created to launch your application such that you should have the following directive in your `*.iss` script:

```
#define MyAppExeName "run.vbs"
```

Next, you should limit what is deployed to just the necessary application files. This should recursively include everything in your `R-portable` directory, but exclude "development cruft" files - i.e. the local git repository, unit test directories, `*.Rproj*` files, etc. You can achieve this via a combination of an `Excludes` directive and a `recursesubdirs` flag in the `[Files]` section. Hence, your `[Files` section should look something like:

```
Source: "C:\path\to\app\*"; DestDir: "{app}"; Excludes: ".git*,.Rproj*,*.Rhistory,te
sts\*"; Flags: ignoreversion recursesubdirs
```

Finally, to allow users to install your application themselves, sans admin privileges, you need to specify "user" locations as opposed to "common" locations like `C:\Program Files\` which tends to be read-only for non-admins:

```
[Setup]
... other setup directives
DefaultDirName={userpf}\{#MyAppName}

...

[Icons]
... other icon directives
Name: "{userdesktop}\{#MyAppName}"; Filename: "{app}\{#MyAppExeName}"; Tasks: deskto
picon
```

The above directives in the `[Setup]` and `[Icons]` sections of the compile script specify that the application will be installed in the current user's Program Files directory (`{localappdata}\Programs`). That said, installing to user locations only works for Windows 7 and higher.

Final thoughts

This is still an application based on modern web standards. Therefore, how interactive/responsive your app can be, and how it looks, will be limited by what can be done with HTML5, CSS, and JavaScript. That said, with the majority of apps, I think a modern web browser should have enough power to provide a good user experience.

What about heavier deployments, like apps that use GTK based UIs via the `RGtk2` package? There are many portable apps that use GTK libraries, so it stands to reason that the toolkit libraries can be packed and deployed with your R application.

Lastly, keep in mind that your source code is easily accessible. If this is a concern for you (e.g. if you are distributing to a client that should not have access to the code) the best you can do is impede access by first compiling the sensitive source code into a binary package. That said, any user who knows R (and has sufficient intent) can simply dump the code to the console.

Overall, I'm very satisfied with this deployment method for R based applications.

Written with StackEdit.

Posted by Lee Pang at 11:41 PM
Labels: app development, programming, r

Franco Peschiera 10 months ago - Shared publicly

I had to change 'shiny::runApp('./shiny/', launch.browser=launch.browser)' to 'shiny::runApp('./shiny', launch.browser=launch.browser)' so R could find the directory.

Also, the app does not show some things when ran this way (my leaflet map

1 · Reply

Lee Pang 10 months ago

Could your leaflet map have a package dependency that you forgot to install in your app's package library?

One way to check would be to launch your app's R-portable console and

Franco Peschiera 10 months ago

Sort of. I realized that I had modified some package (rCharts I think which I also use in the app) in my R installation to make it work. If I copy my installed libraries into the R portable libraries, it works. Thanks.

Edward Kwartler 1 month ago - Shared publicly

I am having trouble using innosetup. Is my "executable" the vbs file and then I also add the entire folder? Can you share your .iss script? thanks.

1

Vaibhav Singh 5 months ago - Shared publicly

Great tutorial. Just had to edit a little for it to work for me.
Made ARGS = sprintf('--apps "%s"', appUrl) as chrome wasn't opening appUrl.
Also Rscript was giving "could not find function loadmethod", didn't work even after adding -e library(moments);
So used R.exe instead.

+2 1 · Reply

Lee Pang 5 months ago

Excellent suggestion for wrapping the app url in quotes. I've also encountered the issue with loadmethod in recent deployments made with R-Portable 3.1.1. I had to add the following to my runShinyApp.R script:

Lee Pang 1 year ago - Shared publicly

Actually, **+ZJ Dai** I used your post as a reference for mine :)

+2 1 · Reply

ZJ Dai 1 year ago - Shared publicly

http://blog.analytixware.com/2014/03/packaging-your-shiny-app-as-windows.html

Oh we posted on a similar topic at around the same time.

+2 1

Jesse Krailler 8 months ago - Shared publicly

I really love this article, but I'm having a little trouble. When I open my batch file, Chrome opens, but it doesn't start the app. Is there something easy that I'm missing? My log file says:

[2260:8516:0828/151122:ERROR:value_store_frontend.cc(62)] Error while

1 · Reply

Lee Pang 8 months ago
Thanks! Regarding your error, have you tested with a system installed Chrome (instead of Chrome Portable)?

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)