



Packaging shiny applications

Maxim Nazarov - Open Analytics

July 12, 2019

What?

Main points

- use **functions** for *UI & Server* components
- use **modules** for application blocks
- **package** everything



How?

Functions for UI & Server components instead of `server.R` and `ui.R` files:

```
myAppUI <- function() {  
  fluidPage( ... )  
}  
myAppServer <- function(input, output, session) {  
  ...  
}
```

Function to launch the application

```
runShinyApp <- function(...) {  
  shinyApp(ui = myAppUI(), server = myAppServer, options = list(...))  
}
```



Why?

Why use functions for UI & Server?

- easier to add arguments for conditional execution, e.g.: *debugging*, bookmarking, different environments, parameterized apps, ...

```
myAppUI <- function(debug = TRUE) {  
  fluidPage(  
    if (isTRUE(debug))  
      actionLink(inputId = "debug", label = "Connect with console")  
  )  
}  
runShinyApp <- function(debug = TRUE, ...) {  
  shinyApp(ui = myAppUI(debug = debug), server = myAppServer, ...)  
}
```



- Belongs together with the other R code (in the package)

Why?

Why use functions for UI & Server?

- easier to add arguments for conditional execution, e.g.: *debugging*, bookmarking, different environments, parameterized apps, ...

```
myAppUI <- function(debug = TRUE) {  
  fluidPage(  
    if (isTRUE(debug))  
      actionLink(inputId = "debug", label = "Connect with console")  
  )  
}  
runShinyApp <- function(debug = TRUE, ...) {  
  shinyApp(ui = myAppUI(debug = debug), server = myAppServer, ...)  
}
```



- Belongs together with the other R code (in the package)

Why?

Why use modules?

- Separate application into logical pieces
- Cleaner code than splitting `server.R` into multiple files and `source()`-ing
- Each module is contained in two functions for *UI* & *Server* components → independent testing possible
- Share and re-use within and between applications



Why?

Why use modules?

- Separate application into logical pieces
- Cleaner code than splitting `server.R` into multiple files and `source()`-ing
- Each module is contained in two functions for *UI & Server* components → independent testing possible
- Share and re-use within and between applications



Why?

Why package shiny applications?

- All the advantages of the R packaging ecosystem:
 - managing dependencies and namespaces (instead of `global.R` file and assorted library calls)
 - versioning, documentation, tests
 - code consistency checks (`R CMD check`)
- Keep application code next to the functional code
 - R code lives in the `R` directory (instead of `inst`)
- Easy to share and distribute



Why?

Why package shiny applications?

- All the advantages of the R packaging ecosystem:
 - managing dependencies and namespaces (instead of `global.R` file and assorted library calls)
 - versioning, documentation, tests
 - code consistency checks (`R CMD check`)
- Keep application code next to the functional code
 - R code lives in the `R` directory (instead of `inst`)
- Easy to share and distribute



Why not?

- May require extra coding/attention in setting up modules and communication between them
- The changes to the *UI & Server* can't be seen without package re-loading / re-installation: this can be facilitated with pkgload/devtools:

```
pkgload::load_all("/path/to/myPackage")  
myPackage::runShinyApp()
```

- Can't use `www` folder in the *UI* function: this can be solved with `system.file` and/or `addResourcePath`



Why not?

- May require extra coding/attention in setting up modules and communication between them
- The changes to the *UI & Server* can't be seen without package re-loading / re-installation: this can be facilitated with pkgload/devtools:

```
pkgload::load_all("/path/to/myPackage")  
myPackage::runShinyApp()
```

- Can't use `www` folder in the *UI* function: this can be solved with `system.file` and/or `addResourcePath`



Where?

ShinyProxy

- All the dependencies are already listed in the DESCRIPTION file of your R package
- Straightforward to create a Docker image:
 - add the R package's .tar.gz
 - install it with `remotes::install_local(..., dependencies = TRUE)`

Shiny server

- When the application folder is required, it can be very minimal: consisting of just an `app.R` file with

```
myPackage::runShinyApp(debug = FALSE)
```



Where?

ShinyProxy

- All the dependencies are already listed in the DESCRIPTION file of your R package
- Straightforward to create a Docker image:
 - add the R package's .tar.gz
 - install it with `remotes::install_local(..., dependencies = TRUE)`

Shiny server

- When the application folder is required, it can be very minimal: consisting of just an `app.R` file with

```
myPackage::runShinyApp(debug = FALSE)
```



Thank you!

Take home:

- See if this approach works for your next project!

Check out:

- shinyproxy.io: enterprise-ready open-source shiny deployment solution

Feedback:

- maxim.nazarov@openanalytics.eu



Thank you!

Take home:

- See if this approach works for your next project!

Check out:

- shinyproxy.io: enterprise-ready open-source shiny deployment solution

Feedback:

- maxim.nazarov@openanalytics.eu

