TL;DR

Learn how to use the ***session*** argument as a global list for passing parameters between the modules in advanced Shiny apps to simplify the objects’ flow in code. ***Session*** can help you organize the app content and simplify the objects flow logic. It is faster than managing all of the dependencies between modules manually.

This is the first in a 5 part series of tips and tricks for advanced Shiny developers.  The following tips will help keep your Shiny apps bug-free, transparent, organized, fast, and reliable.

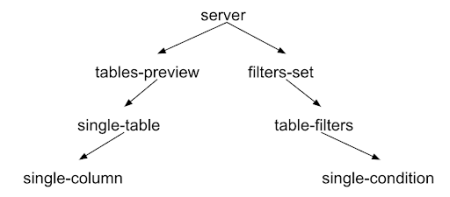
**Introduction**

When going through “hello world!” applications one may not even consider “**how to pass values between modularized applications**” as a potential issue. However, most advanced Shiny practitioners bump into this problem when creating multi-view, modularized applications.

**Context & the Problem**

What are the roots of the problem? Shiny was designed for simple prototype applications used by data scientists to help in everyday work. However, the community quickly discovered that Shiny’s potential is far greater and started to develop advanced dashboards. Keeping the whole app structure in just two files, *ui* and *server,* becomes less effective after a certain number of lines of code. What is more, once you build an extensive part of the application you would like to re-use it in your other apps. And that is when the Shiny modules idea comes on the stage – developers can now build the parts of the code in separated files and mix them together into advanced dashboards. Great! However, there is a challenge here: how can the separated parts be interdependent inside one app yet independent enough to be used elsewhere?

As an app grows bigger and its modules’ dependencies become more complicated, it gets harder to organize all the traffic between saved results and parameters passed to modules. Let’s work on a real life example of nested modules: there is a screen (1st tier module) with a few tables (single table is 2nd tier module) and each table consists of a few columns (column is 3rd tier module). There is also a module with filters (e.g. opened on modal) with sections for each table and with module for a single condition.



Using a basic Shiny system requires passing the parameters up-and-down the dependency ladder. Imagine that you need to use yet another parameter from *single-condition* in *single-column*. That might require a lot of changes in parameters passed in each module! Don’t get me wrong – it will work fine, and if organized nicely it might not even be a mess. Nevertheless using the *session* solution will bother you much less. Let’s have a look at the details.

**The Solution**

As you probably know, the *ui* part of the module should always contain at least the *id* parameter and *server* has those boilerplates parameters *input, output, session*. As each module has access to *session,* anything that we store there is accessible in each module. You can treat it as a global list of stored values. You may be familiar with *ns <- session$ns,* which is the same concept. Now we will use it for passing the other values as well. One note: *session* needs to be added as an argument to the main *server* function; it might be useful here especially when you would like to save some user data when the app starts.

In this rather simple app the module *scatterplot\_server\_mod* requires three additional parameters (*dataset, plot1vars, plot2vars*) to be passed to the module call. We will try to simplify it and keep the functionality working without using external parameters. It might seem like overkill in this case, but as we see in our example, things can escalate quickly.

All we need to do is save the results in *server* file not in local objects, but into the list stored in global session. Let’s turn this:

# server logic

server <- function(input, output, session) {

# prepare dataset

ames <- make\_ames()

# execute plot variable selection modules

plot1vars <- callModule(varselect\_mod\_server, "plot1\_vars")

plot2vars <- callModule(varselect\_mod\_server, "plot2\_vars")

# execute scatterplot module

res <- callModule(scatterplot\_mod\_server,

"plots",

dataset = ames,

plot1vars = plot1vars,

plot2vars = plot2vars)

}

Into this:

# server logic

server <- function(input, output, session) {

# prepare dataset

session$userData$dataset <- make\_ames()

# execute plot variable selection modules

session$userData$plot1vars <- callModule(varselect\_mod\_server, "plot1\_vars")

session$userData$plot2vars <- callModule(varselect\_mod\_server, "plot2\_vars")

res <- callModule(scatterplot\_mod\_server, "plots")

}

And now in the module we can use the values from *session* as we wish. For simplification it can be assigned to the same variables and the rest of the code will stay untouched:

# assign just not to use whole object all the time

plot1vars <- session$userData$plot1vars

plot2vars <- session$userData$plot2vars

dataset <- session$userData$dataset

The beauty of the solution is that if we call the module in some different place e.g. nested in other module on separated modal screen we do not need to care whether the variables will be accessible for it or if re-calling them is needed. They are always there waiting for us on the *session*!

One note: you’ve probably realized that it violates modules independence as they use external objects without stating them explicitly as parameters (well, technically *session* is a parameter, but you get the point). Nevertheless, the practice of having them re-assigned to the simpler objects at the beginning of the module assures that you won’t miss this dependency. Also please remember that it is advised not to keep too much data in the single user session.

To sum up: keeping the results of the modules in a *session* object will help you organize the app content and simplify the objects flow logic. It is faster than managing all of the dependencies between modules. We recommend this approach for advanced Shiny apps.