A Brief Cookbook for Module Writers

Zhengjia Wang

2017-11-07

## Package Structure

It's important to know the structure of RAVE before writing a module. Since ECoG data is big, and we implement our package in a way that all the data is loaded into memory (though new format is under test for computers with small RAM, it's still recommeded that those computers upgrade RAM), we design the structure that seperate data repository with modules, which brings the following advantages:

* Data is shared across the sessions.
  + For example, if you open two browser tabs (running on same RAVE instance), entering the IP address of the instance running. You can set different parameters for each sessions and compare the results. However, since they are running on one RAVE instance, RAM shouldn't change too much.
* Data is be shared across the modules.
  + No need to import data each time a new module is opened.
  + Loading data doesn't require shiny, therefore we can perform analysis/writing modules without creating RAVE instances. This feature will be introduced later
  + With same format for different subjects, module writers can focus on implementing visualizations.

We are not going any further into the implementation. However, just remember that modules are session-based, and they are separated from data. When writing modules, we can always assume that the data is loaded. Therefore it's important to know what information is loaded before adding modules.

Let's load a subject without lauching any browsers. This step helps us understand setting up develop environment. However, it should never appear in the module scripts since RAVE will automatically run the code for us.

# Play with a toy dataset  
  
# STEP 1: Set options  
rave\_opts$set\_options(  
 data\_dir = system.file('example/data', package = 'rave')  
) -> opt; # Assign to avoid printing  
  
  
# STEP 2: Load subject  
find\_subject\_ids() # List all subject IDs

## [1] "dipterix"

subject\_id = 'dipterix' # Let's use the first ID as an example  
  
subject = get\_subject(subject\_id) # Load subject meta-data  
  
  
# STEP 3: Load ECoG value, for example, first four  
subject$electrode\_label\_by\_index(1:4)

## [1] "N1" "N2" "N3" "N4"

suppressMessages({ # Suppress loading messages.  
 subject$data\_environment$load(electrodes = 1:4) # Prepare electrodes  
 subject$data\_environment$bind\_electrodes(electrodes = 1:4, debug = T) # Concatenate tensors  
})

## [DEBUG]: Try to load from cache...  
## [DEBUG]: Try to load from cache...  
## [DEBUG]: Try to load from cache...  
## [DEBUG]: Try to load from cache...  
## [DEBUG]: Try to load from cache...  
## [DEBUG]: Try to load from cache...  
## [DEBUG]: Try to load from cache...  
## [DEBUG]: Try to load from cache...

## NULL

# Check data  
data\_env = subject$data\_environment  
dim(data\_env$data) # Trial x Freq x Time x Electrode (Valid)

## [1] 4 10 51 4

rm(subject, data\_env)

During module development, we use function attach\_virtualenv to load subject data. In this case, we can also use attach\_virtualenv(subject\_id, 1:4) after step 1 to get data\_env. However, attach\_virtualenv hides some details.

We can devide the code above into three steps as follows:

1. Setting up option, or creating configuration file
2. Load a subject's meta data into memory
3. Load electrodes needed for the analysis

(*Read vignette A Cookbook for RAVE Users for details in changing settings and preparing subject data.*)

Here we can see that in order to load data environment, RAVE applies two steps. First, loading subjects' meta information, which is in [subject]/ecog/meta, the descriptions of trials, frequencies, time points and electrodes. Then, according to user's requirement, load electrodes of interest.

The result in this case, data\_env, has one element data, which is a four-mode tensor: trialxfrequencyxtimexelectrode. If we apply attach\_virtualenv, we can see other elements such as cumsum, electrodes, etc.

attach\_virtualenv(  
 subject\_id = subject\_id,  
 electrodes = 1:4  
)

## Virtual environment created:  
## Here are variables you might want to use for development

## [1] "data\_env" "get\_global\_var"

## Here's what you can find in data\_env:

## [1] "cumsum" "data" "electrodes" "subject"

## For example, data\_env$data contains ecog tensor data

## while data\_env$subject contains subject info

## --- Type detach\_virtualenv() to quit this environment. Enjoy :)

ls(data\_env)

## [1] "cumsum" "data" "electrodes" "subject"

* data\_env$data: four-mode tensor
* data\_env$cumsum: four-mode tensor, cumulative summation of data\_env$data over time
* data\_env$electrodes: current loaded electrodes
* data\_env$subject: subject object that contains meta data
* get\_global\_var: see next part

Now we have created environment for writing modules.

## Writing the first module

### Structure of a basic module

To write our first module, here's what we have:

data\_env$data, data\_env$cumsum, data\_env$electrodes, data\_env$subject

These data are **read-only**. Never change them. Also, we don't recommend assignments such as mydata <- data\_env$data since it will copy the data, which could be as large as 10GB, in the memory.

To make it a module, here's what we need to implement:

SHINY\_INPUT, SHINY\_EXECUTE, SHINY\_OUTPUT

There are some other variables that are optional:

SHINY\_DESCRIPTION

Please copy this template save save it as a example.R file.

###########################  
# Example module  
# Author: Zhengjia Wang  
###########################  
  
  
# Definitions of input  
# There are more than seven types of inputs, for "text\_input", type "?text\_input" to see documentations  
#  
# For all inputs, "inputId" are important since they are local names for your inputs  
# For example, "text\_input(inputId = 'textA',..."  
# Here we have a variable "params$textA" that you can use as a \*string\* in `SHINY\_EXECUTE`  
SHINY\_INPUT = list(  
 text\_input(inputId = 'textA', label = 'Please enter a text', init = function(){  
 return(list(  
 value = get\_local\_var('textA', paste('Default text', Sys.time()))  
 ))  
 }),  
 numeric\_input(inputId = 'numB', label = 'This is a random number', value = 0, init = function(){  
 return(list(  
 value = rnorm(1)  
 ))  
 })  
)  
  
# (optional) parameters for local debug  
# We have two input IDs, "textA", and "numB"  
params = list(  
 textA = 'debug text',  
 numB = 1  
)  
  
# Algorithm part, process data here  
SHINY\_EXECUTE = function(params, ...){  
   
 # Pre-process data  
   
 s = paste(params$textA, '|', params$numB)  
   
 # Return a named list of functions  
 return(list(  
   
 # The name "output\_text" will be used as output ID in "SHINY\_OUTPUT"  
 output\_text = function(){  
 return(s)  
 }  
 ))  
}  
  
# Visualization settings, a named list of output tabs and components  
SHINY\_OUTPUT = list(  
   
 # "First tab" is the name of the tab  
 `First tab` = list(  
   
 # "output\_text" is output ID defined in "SHINY\_EXECUTE"  
 verbatimtext\_output(outputId = 'output\_text', title = 'Output Text')  
 ),  
 `Second tab` = list()  
)

This is our first module. It will verbatim a text which concatenate the two inputs. However, in order to put them in RAVE, we need two more steps:

1. Create a module index csv file (for example, modules.csv), and enter/append the following table. Check system.file('modules.csv', package = 'rave') for example

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| category | module\_id | label | source\_path | active | packages | author | version | is\_univariate | suma\_enabled | order |
| Test | example\_module | First Module | C:/Users/zheng/Documents/R/win-library/3.4/rafe/example/module/example\_module.R | TRUE | stringr | Zhengjia Wang | 0.0.1 | TRUE | TRUE | 100 |

1. Register index file modules.csv to RAVE:

rave\_opts$set\_options(  
 module\_lookup\_file =  
 'C:/Users/zheng/Documents/R/win-library/3.4/rave/example/module/modules.csv'  
)

Now let's lauch the web UI

# Set options,  
# Specify data directory  
# Specify module index file  
#   
rave\_opts$set\_options(  
 data\_dir = system.file('example/data', package = 'rave'),  
 module\_lookup\_file = system.file('example/module/modules.csv', package = 'rave')  
) -> opt;  
  
# Command to lauch app  
init\_app()

### Inputs

SHINY\_INPUT is a list of input components. Here is the list for inputs:

|  |  |
| --- | --- |
| Component | Type |
| text\_input | Characters |
| numeric\_input | Number |
| checkbox\_input | Boolean |
| select\_input | Characters |
| slider\_input | Number(s) |
| action\_button | Signal |
| file\_input | File Table |

Usage:

text\_input(inputId, label, init, global\_var, value, width, placeholder)  
numeric\_input(inputId, label, init, global\_var, value, min, max, step, width)  
checkbox\_input(inputId, label, init, global\_var, value, width)  
select\_input(inputId, label, init, global\_var, choices, selected, multiple, width, size)  
slider\_input(inputId, label, init, global\_var, min, max, value,   
 step, round, format, locale, ticks, width, sep, pre,  
 post, timeFormat,timezone, dragRange)  
action\_button(inputId, label, init, global\_var, icon, width)  
file\_input(inputId, label, init, global\_var, multiple, accept,  
 width, buttonLabel, placeholder)

* inputId: (Mandatory) variable name, will be stored in params and called as params$inputId or params[['inputId']]
* label: (Mandatory) text that will be displayed in app
* value/selected: (some are optional), default value for components
* init: (optional) a function with no arguments and returns named list of initial/default values for components
* global\_var: (optional) name for the component if this input will be used in other modules. NULL by default, meaning that the component can only be accessed by local module.
* width: (optional) width of the component, usually ignore this argument *(Type "?" for documentation, for example, ?text\_input)*

### Data process

SHINY\_EXECUTE is the body part that we process our data with inputs. The format follows:

1 SHINY\_EXECUTE = function(params, ...){  
2 [YOUR\_CODE\_HERE]  
3   
4 return(list(  
5 [OUTPUT\_ID\_1] = function(){  
6 [CODE\_GENERATING\_GRAPH/TABLE]  
7 },  
8 [OUTPUT\_ID\_2] = function(){  
9 ...  
10 },  
11 ...  
12 ))  
13 }

Line 1 is fixed. We can see that SHINY\_EXECUTE itself is a R function, which takes params and ... as arguments (... is a special keyword, reserved for future development).

params is a list, with number of elements equaling to the number of inputs, and their names are input IDs. For instance, if SHINY\_INPUT has a component with inputId='textinput', then we can access users' inputs for this component via params$textinput or params[['textinput']] within SHINY\_EXECUTE.

The output of SHINY\_EXECUTE should be a list of functions, with output ID as the name for each functions. For example, if we want to have a plot output with outputId='myplot' which plots Sin(x) in the SHINY\_OUTPUT, then we need to have SHINY\_EXECUTE returns a list as the example below:

[WITHIN\_SHINY\_EXECUTE]  
...  
return(list(  
 myplot = function(){  
 plot.function(sin)  
 }  
))  
[END\_OF\_SHINY\_EXECUTE]

Make sure that myplot is a function that takes no arguments.

### Outputs

SHINY\_OUTPUT defines output. The structure of SHINY\_OUTPUT is defined as a list of tabs, with each tab containing output components.

tabcomponets\_1 = list(  
 [Output Component 1],  
 [Output Component 2],  
 ...  
)  
  
...  
  
SHINY\_OUTPUT = list(  
 [Tab name 1] = tabcomponets\_1,  
 [Tab name 2] = tabcomponets\_2,  
 ...  
)

The output will be displayed in tabs, with "[Tab name 1]", "[Tab name 2]", etc. Within each tabs ("tabcomponets\_1", "tabcomponets\_2", ...), there should be a list of components. Similar to the inputs, there are four types of outputs:

verbatimtext\_output, datatable\_output, plot\_output, plotly\_output

Usage:

\*\*\*\_output(outputId, title, ...)

* outputId: matches with names of function returned by SHINY\_EXECUTE
* title: title for output to be displayed

Here's an example for plots

SHINY\_EXECUTE = function(params, ...){  
 ...  
   
 return(list(  
 plotA = function(){  
 plot.function(sin)  
 }  
 ))  
}  
  
SHINY\_OUTPUT = list(  
 Visualization = list(  
 plot\_output(outputId = 'plotA', title = 'Curve for Sin(x)')  
 )  
)