Adding Shiny Modules

Jenna Reps, Jamie Gilbert, Josh Ide

2022-07-05

Contents

1 Introduction

A shiny module consists of two parts: the user interface (UI) function and the server function. The UI function contains the code that specifies what inputs/outputs are seen when the module is called and the server function contains the code to do any process required by the module. If an input is dynamic (e.g., it depends on some user interaction) then it may need to be specified in the server rather than the UI.

2 What Is A Shiny Module

Both the UI function and server function require the input `id` that is a string and much match between the UI and server for the same module instance. For a nice tutorial on shiny modules see this website.

2.1 UI module function

The module's UI function specifies how the shiny module will be displayed to the user. The string variable `id` should be an input for every model UI as it is used to create the namespace. The main aspect of a UI module is the namespace function `ns <- shiny::NS(id)`. This is put as the first line of any module UI function. Effectively, what this function does is append the id string to the input of the `ns()` function. For example, if a user calls the UI module by running `exampleViewer(id = 'example_name')` then the namespace function `ns()` will concatenate the string 'example_name' to every reference in the UI, e.g., `ns('serverReference')`

= 'example_name_serverReference'. Therefore, as long as the id input values are unique for each instance of a module, the references within the modules viewer and server will be unique and not clash across modules.

```
exampleViewer <- function(id) {
  ns <- shiny::NS(id)

  shiny::fluidRow(

    shiny::column(width = 12,

                  shinydashboard::box(
                    width = 12,
                    title = "Example Title ",
                    status = "info",
                    solidHeader = TRUE,
                    DT::dataTableOutput(ns('serverReference'))
                  )
    )
  )
}
```

2.2 Server module function

The module's server function is where all the data fetching and manipulation happens. The module server and UI server is linked by using the same id for example, when calling the server you would use exampleServer(id = 'example_name'). For the example above, the server function needs to specify what the serverReference dataTable is. The server automatically knows the namespace, so the DT::dataTableOutput(ns('serverReference')) can be define using the output output$serverReference (no namespace fucntion is require in the server code).

```
exampleServer <- function(
  id,
  extraInput
  ) {
  shiny::moduleServer(
    id,
    function(input, output, session) {

      output$serverReference <- data.frame(a=1:5, b=1:5)

    }
  )
}
```

2.3 Calling the modules in the main UI and server

```
ui <- shiny::fluidPage(
  exampleViewer("exampleName")
)
server <- function(input, output, session) {
  exampleServer("exampleName")
}
shiny::shinyApp(ui, server)
```

2.4 Useful shiny packages

The following packages are useful for developing shiny modules:

- reactable: contains a nice shiny table
- plotly: used to create interactive plots
- ggplot: used to create static plots

3 Guidelines for contributing a module

If you would like to add a new module please ensure you follow the following guidelines:

3.1 Style

We recommend following the HADES coding style details here.

3.2 File names

Please name files in the following format: The main module should be called `<module name>-main.R` and the submodules within that module should be called `<module name>-<submodule name>.R` for example the main prediction module is called `prediction-main.R` and the discrimination sub module within the prediction module is called `prediction-discrimination.R`.

3.3 Help/Information buttons

Please place any html or markdown files inside the package 'inst' folder. The helper information about the module should be inside the subfolder '-document' and any helper files used within the module should be in '-www'. The location of these files can be determine within the module using the following code:

```
system.file('<module name>-www', file, package = "OhdsiShinyModules")
```

See `R/helpers-getHelp.R` to see any example of a function used by the prediction module to get the helper file locations.

3.4 Testing

A new module must have >80\% test coverage before it is added to the package. Testing shiny modules is similar to testing R packages with testthat, however, there are some notable specific differences for this set up that must be followed.

Testing of module servers can be done using the `shiny::testServer()` function. For an example of how to write shiny tests please see `/test/testthat/test-prediction-main.R`.

3.4.1 Creating test files

In the test directory test files should have the prefix `test-*.R`. Naming convention for test files is: `test-<module name>-<submodule>.R` and the start of the file should use the `context()` function.

For resued fixtures and useful functions in tests, you can also include the files `tests/helper-objects.R` (generic objects) `tests/helper-<module name>.R` (objects specific to the module) which will be loaded when the tests start.

For example, database connection strings should be included in `tests/helper-objects.R`.

Tests can follow the normal testthat patterns, however, testing a shiny module requires loading a shiny test server that can be used to simulate user inputs and test the behaviour of `shiny::reactive` calls as well as outputs.

For example:

```
test_that("Example Test", {
  shiny::testServer(exampleModuleServer, args = list(id = "testModule"), {

    # session$setInputs allows simulation of user behaviour
    session$setInputs(
      testString = "foo"
    )

    # after modifying input output can be verified
    expect_equal(myReactive(), "Input is foo")
    expect_equal(output$testOutput, "Input is foo")

  })

})
```

3.4.2 Test data and fixtures

The modules themselves will also require test data. For that reason it is recommended that an sqlite database containing test data should either be created in `helper-<module name..R` or be stored as flat files in the `tests/resources` folder. For example, currently in `tests/resources` you'll see the file 'databaseFile.sqlite' that contains example prediction result data generated using the Eunomia package. This is used to test the prediction module. Please place similar files for different analyses types in this folder.

3.4.3 Global setup files

The following `tests/helper-objects.R` file loads the required database connection string:

```
# setup.R
connectionDetails <- DatabaseConnector::createConnectionDetails(
  dbms = 'sqlite',
  server = 'testDb.sqlite'
  )
```

Tip 1: the file `tests/testthat/helper-objects.R` contains R test code that gets executed before any other test. This is a useful place to run any generic code that may be reused in multiple test files.

3.4.4 Running tests

These tests can then be called with the testthat command:

```
testthat::test_dir('<path_to_package>/tests')
```

3.5 Dependencies

The current dependencies include the standard shiny packages, plotly, ggplot and a selection of helper OHDSI packages. Ideally new modules should use the existing dependencies. If a new dependency is required please post in the GitHub issues the required dependency so there can be a discussion whether to add the dependency or 'borrow' the specific code required by the module.