November 2024

R Shiny Masterclass Series - Advanced

Session 4

Managing complexity: modularising code with the module pattern

EPI-interactive

Agenda

Session 1 | October 30th | Introduction, recap and responsive interfaces in R Shiny

Session 2 | November 1st | Advanced reactivity

Session 3 | November 5th | Useful R packages to extend core Shiny functionality

Session 4 | November 6th | Managing complexity: modularizing with the module pattern

Session 5 | November 8th | Advanced data sources and processing

Session 6 | November 12th | Automated report generation

Session 7 | November 13th | User authentication, Extended exercise

Session 8 | November 15th | AI Tools, Programming sins and how to avoid them

Today

- Shiny.router
- Modules
- Modularising an existing Shiny app
- Getting data from modules

Routing your app with shiny.router

Tabs vs Routing

Tabs

- Single page application
- Fewer but larger files
- Indirect use of URL search to anchor tabs (configuration needed)

Routing

- Meaningful URLs
- Smaller, decoupled files
- User can easily go to a specific part of the application

shiny.router

- https://appsilon.github.io/shiny.router/
- Easy interaction with R Shiny modules
- Need to design applications with shiny.router in mind
- Can use back/forward in browser

- Can either navigate using regular hyperlinks, or in the server with the change_page() function

Shiny router: server.R & ui.R

server.R

```
router_server(root_page = “/”)
```

Ui.R

```
tags$ul(
     tags$li(a(href = "#!/", "Home")),
     tags$li(a(href = "#!/other", "Other"))
),
router_ui(
    route(“/”, fluidPage(“Home page content”)),
    route(“other”, fluidPage(“Other page content”)
)
```

EPI-interactive

Modules

Our current file structure

- ui.R, server.R, global.R
- All layout and functionality grouped together
- This does not scale well, apps can get very large codebases very quickly, becoming:
  - Disorganised
  - Difficult to read
  - Inefficient
  - “Monolithic”

Modules

- Organise recurring functionality into a single component for easy re-use
- Group related elements / functionality together
- Modules should have a single purpose

- UI components become a function to generate the UI elements
- Server logic becomes a function, with **moduleServer**

Using Modules

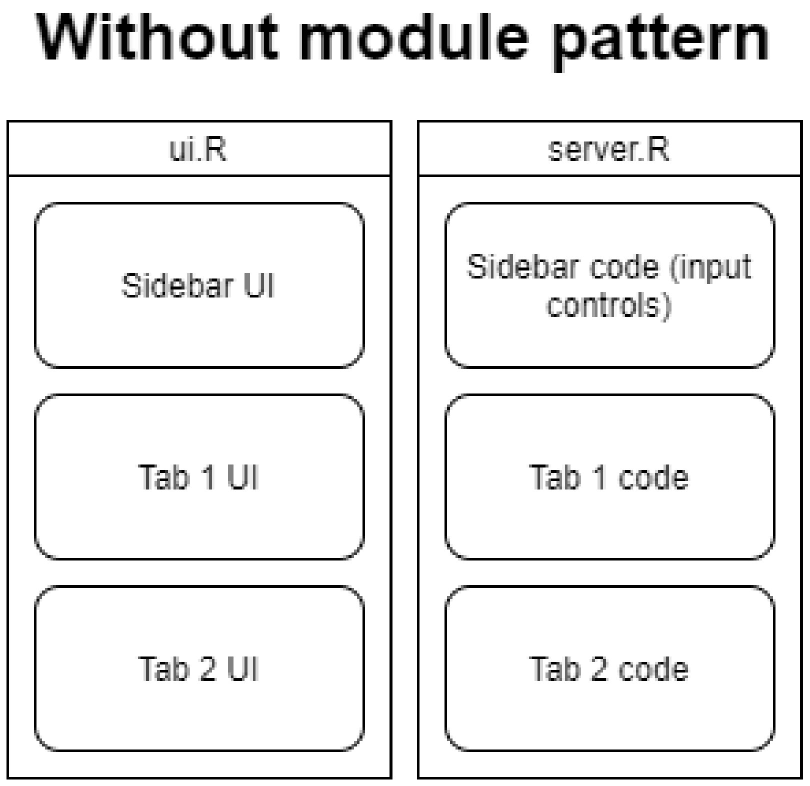
Why?

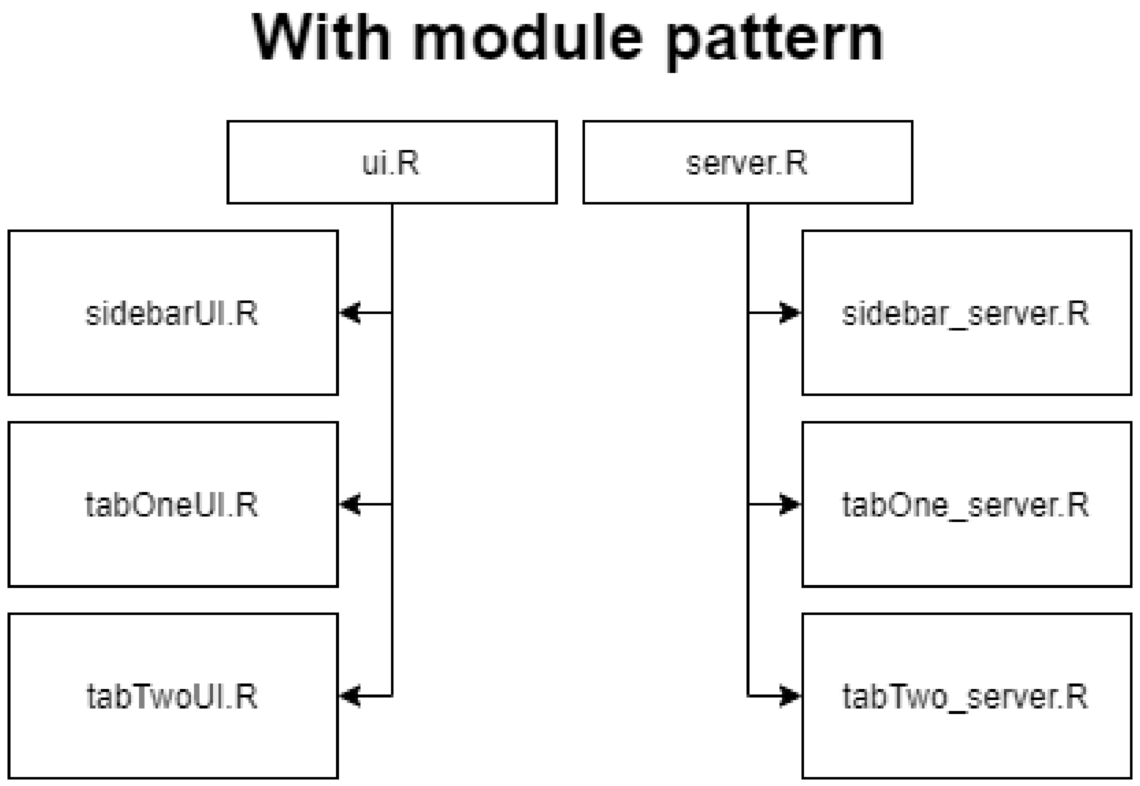
- Code reuse
- Only need to make changes once
- Easier to read
- Smaller files
- Easier to navigate

How?

- Page-specific modules
- Tab-specific modules
- Component specific modules
  - Sidebar controls
  - Selection modal
    (see GitHub for details)

Using modules



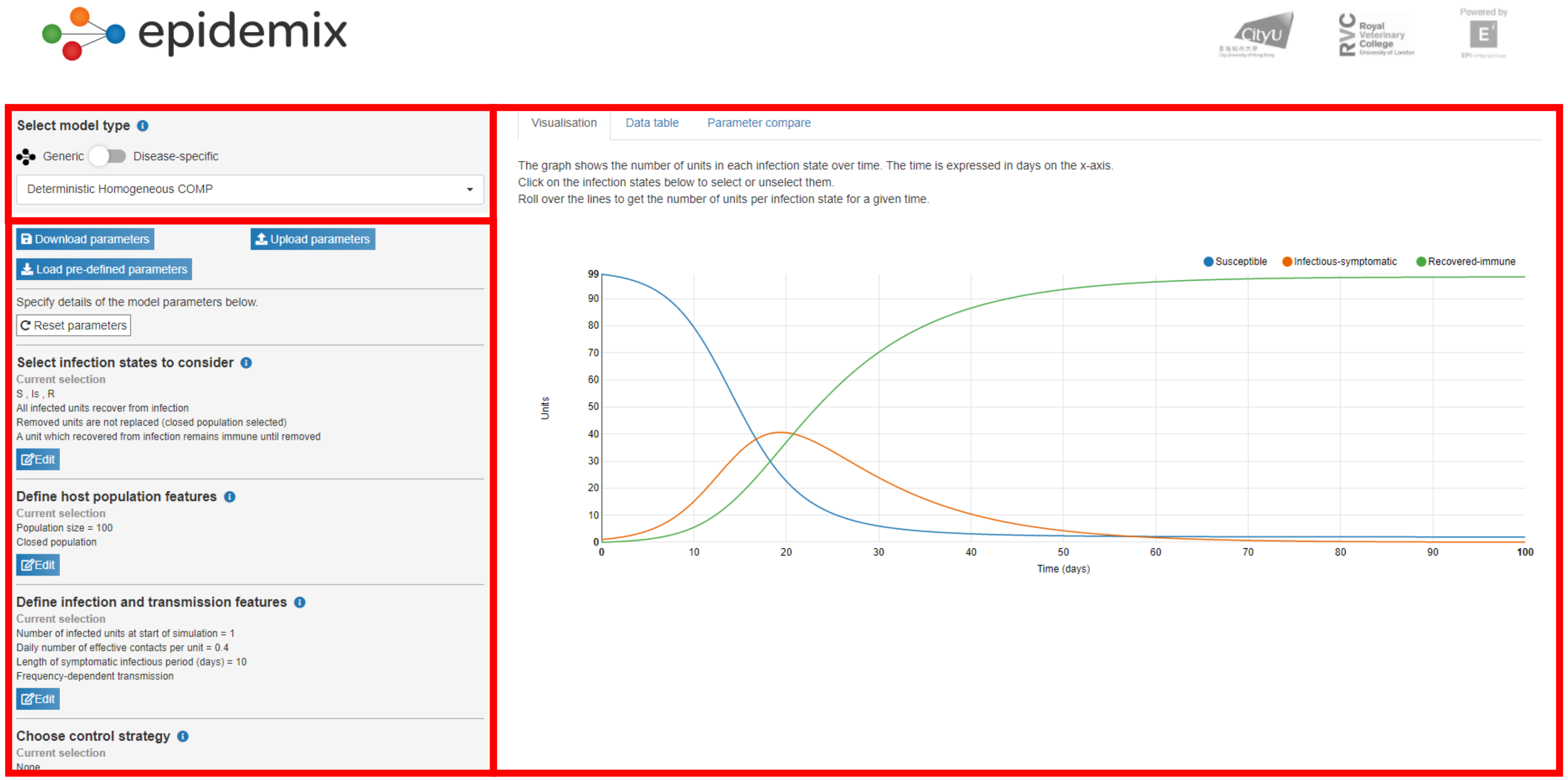


Using modules

Points to consider:

- Does it really need to be a module?
  - Is your app large enough for modules to be useful
  - Is this functionality large / complex enough to be a module
- Are there too many independent variables?
- Will the code be used more than once as a module?
- How does this functionality link to the rest of your app?

EXAMPLE: Epidemix



epidemix

Powered by

Select model type

Generic | Disease-specific

Deterministic Homogeneous COMP

Download parameters | Upload parameters | Load pre-defined parameters

Specify details of the model parameters below.

Reset parameters

Select infection states to consider

Current selection
S , Is , R
All infected units recover from infection
Removed units are not replaced (closed population selected)
A unit which recovered from infection remains immune until removed

Edit

Define host population features

Current selection
Population size = 100
Closed population

Edit

Define infection and transmission features

Current selection
Number of infected units at start of simulation = 1
Daily number of effective contacts per unit = 0.4
Length of symptomatic infectious period (days) = 10
Frequency-dependent transmission

Edit

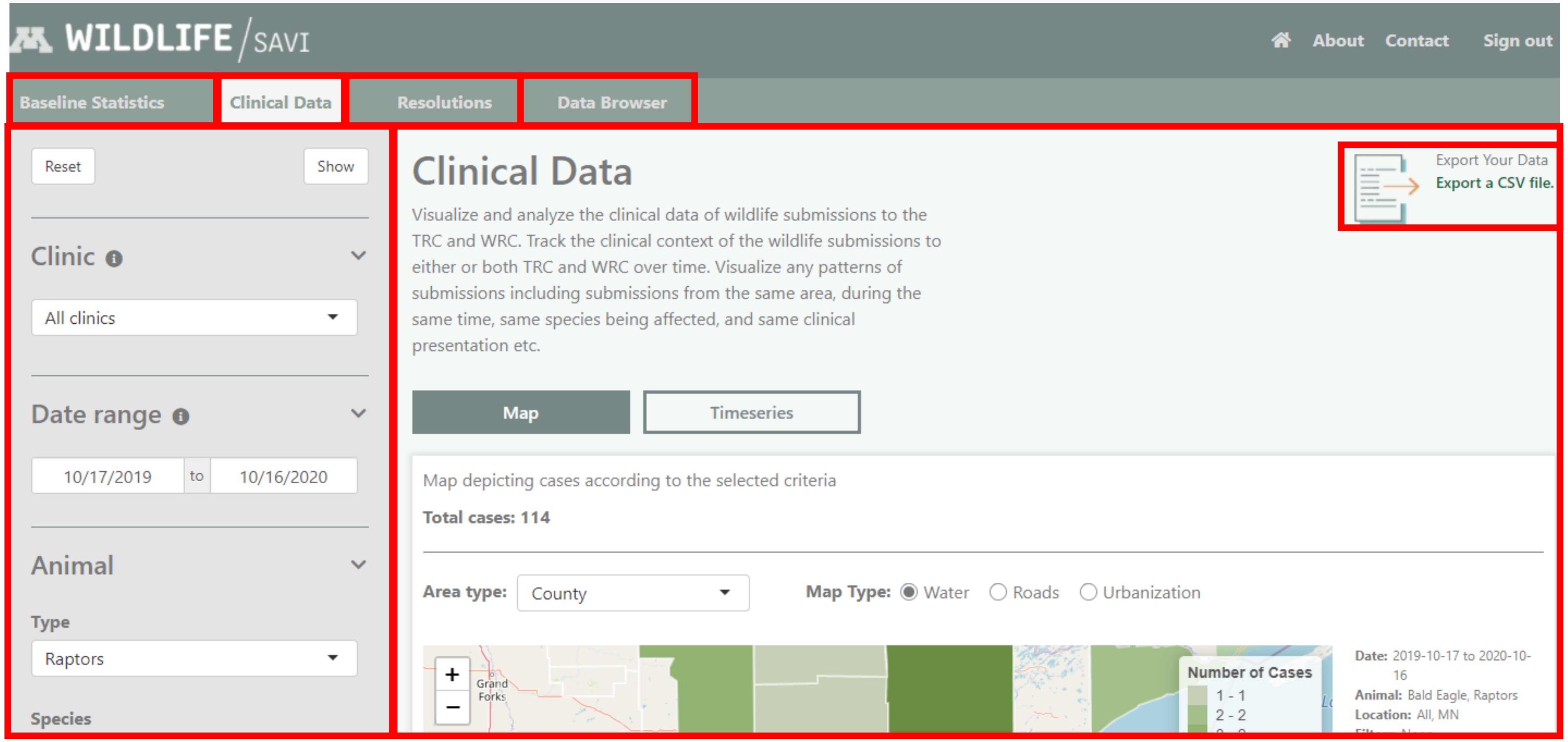
Choose control strategy

Current selection
None

Visualisation | Data table | Parameter compare

The graph shows the number of units in each infection state over time. The time is expressed in days on the x-axis.
Click on the infection states below to select or unselect them.
Roll over the lines to get the number of units per infection state for a given time.

EXAMPLE:
UMN Wildlife


WILDLIFE / SAVI

About · Contact · Sign out

Baseline Statistics | Clinical Data | Resolutions | Data Browser

Reset · Show

Clinic: All clinics

Date range: 10/17/2019 to 10/16/2020

Animal

Type: Raptors

Species

Clinical Data

Visualize and analyze the clinical data of wildlife submissions to the TRC and WRC. Track the clinical context of the wildlife submissions to either or both TRC and WRC over time. Visualize any patterns of submissions including submissions from the same area, during the same time, same species being affected, and same clinical presentation etc.

Export Your Data
Export a CSV file.

Map | Timeseries

Map depicting cases according to the selected criteria

Total cases: 114

Area type: County **Map Type:** [✓ Water] Roads Urbanization

Date: 2019-10-17 to 2020-10-16
Animal: Bald Eagle, Raptors
Location: All, MN

Namespaces

- Shiny apps use **unique** IDs to identify input/output elements
- Collisions between these IDs causes elements to break
- We use namespaces to avoid these collisions

ns <- NS(tag, id = NULL) to create the namespace

ns(“id”) to create the unique ID

Example:

- NS(tag = “overview”, id = “plot”), or ns(“plot”) becomes “overview-plot”

Namespaces in modules

- Only need to use when using modules
- Server: use session$ns.
- UI: We create a Namespace object first, then use that to create our IDs
- Need to use wherever we ***create*** an inputId/OutputId:
  - inputId = “slider” becomes inputId = ns(“slider”)
  - outputId = “chart” becomes outputId = ns(“chart”)

UI / server for each module **MUST** have the same namespace!

Module: UI function

```
overview_UI <- function(id) {
    ns <- NS(id)
    fluidRow(
        column(3, uiOutput(ns(“sidebar”))),
        column(9, uiOutput(ns(“mainContainer"))))
    )
}
```

- Needs to return a single container UI object (E.g. div, tagList, fluidRow)

EPI-interactive

Module: replacing ui.R

```
ui <- fluidPage(
    fluidRow(h1(“Page title”)),
    overview_UI(“overview”),
    …other page content here…
)
```

- Just need to call the UI function in the appropriate place for your application.
- Exact details depend on the content inside your UI function

Module: server function

```
pageOverview <- function(id) {
      moduleServer(
            id,
            function(input, output, session) {
                  ns <- session$ns
                  output$mainContainer <- renderUI({
                        …
                  })
            }
      }
```

- Added session parameter to server function. This allows us to use the namespace function ***ns***

Module: replacing server.R

- As a stand-alone block of functionality:

```
server <- function(input, output, session) {
module_Server(“overview”)
}
```

- Nested inside other functionality with a return value:

```
server <- function(input, output, session) {
    someContent <- reactive({
        content <- module_Server(“overview”)
        …processing the content here…
    })
}
```

EPI-interactive

Modules with Shiny.router

- The way we use modules fits Shiny.router functionality well
- Page level modules a common approach
- Call module UI in a route, call server module as normal*

Example:

```
# in ui.R
router_ui(
      route(“index", index_UI("index"))
)

# in server.R
index_Server("index")
router_server(root_page = “index”)
```

EPI-interactive

Module based file structure

- Good practice to separate modules into their own UI/Server files
- Need to load these files before we can call the modules
- Example with routing:
  - Server.R
    - pageOverview("overview“)
  - Global.R
    - source("source/pages/overview.R")
    - source("source/pages/overview_server.R")
    - route("overview", pageOverviewUI("overview"))

Module based file structure

- Ui.R
- Server.R
- Global.R

Becomes

- Ui.R
- Server.R
- Global.R
- Source
    - Modules
        - module_UI.R
        - module_Server.R

Parameterised modules

- Ideally, we want to keep our modules as self-contained as possible
- Some scenarios where we might need to pass data into / out of a module:
  - Parameters when calling module
  - Global variables
  - Reactives
- Can return data from a module in a reactive, treat module like a reactive itself

Parameterised modules: example

```
example_Server <- function(id, other_param) {
    moduleServer(
        id,
        function(input, output, session) {
            ns <- session$ns
            dat <- reactive({
                out <- other_param %>% … # some data processing here
            })
        return(dat)
    })
}

data <- example_Server(“index”)
```

EPI-interactive

Modules: things to watch out for

- Make sure that paired UI & Server modules both use the same namespace
- Once called, a module can’t be disabled (goes until application ends)
- **Every** id declared inside of a module UI or server needs to have the appropriate namespace applied
- No circular dependencies

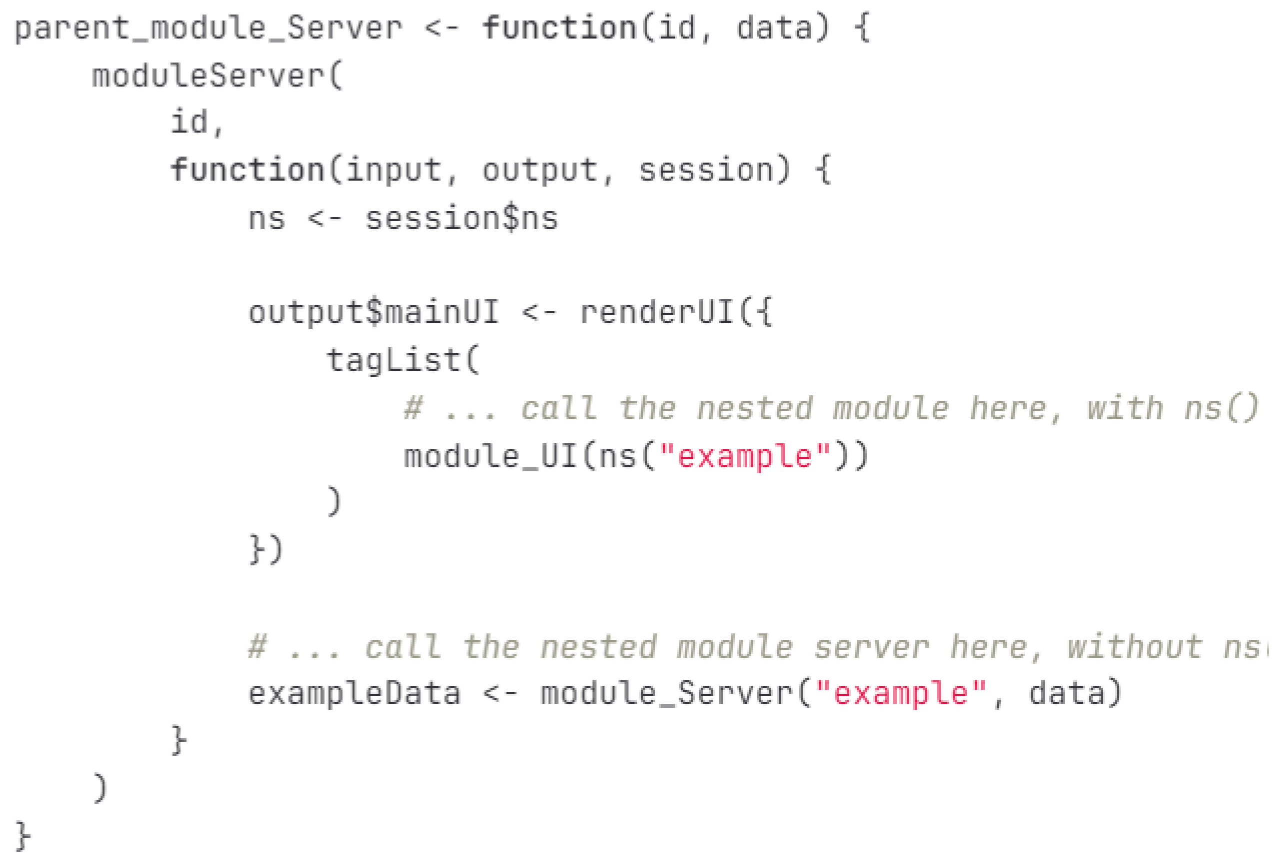
Exercise

Using /stage1, let’s modularise and fix up this Shiny app!

- Create new files for two new modules – one for the chart tab and one for the table tab. Source these new files in global.R
- Define the UI and server functions for these two modules and set up the namespaces
  - *Hint: look at the IDs created in nav-module.R, ensure these match up!*
- Move the content over from ui.R and server.R to the appropriate module file, and ensure they are namespaced correctly
- Modify the existing filters module to return the filtered_data reactive, then ensure that this is being passed to your new chart / table modules
- Call these modules in ui.R and server.R to complete the set-up
  - *Hint: look for the shiny.router call in ui.R, and the function calls for nav_server and filters_server as examples*

Nested modules

- It is possible to call a module from inside another module!
- In this case, the namespaces are combined
- Slightly different behaviour for calling the ui / server parts:



```r
parent_module_Server <- function(id, data) {
    moduleServer(
        id,
        function(input, output, session) {
            ns <- session$ns

            output$mainUI <- renderUI({
                tagList(
                    # ... call the nested module here, with ns()
                    module_UI(ns("example"))
                )
            })

            # ... call the nested module server here, without ns
            exampleData <- module_Server("example", data)
        }
    )
}
```

EPI-interactive

Next time

- Advanced data sources and performance

Challenge (using /result):

- Create a new page using the table module, called 'region summary' (I.e. do not create a new module, but re-use your existing module with a new id)
- Pass as a parameter to this module a reactive, containing all countries for the specified region(s)
- Create a new module to handle downloading data from the application and add this as a sub-module of the table page module. This should have:
  - In the ui, a downloadButton
  - In the server, accepts an additional parameter for data, and has a downloadHandler to export a CSV file

- Share your result on the session 4 forum!