# 1. Logic of App

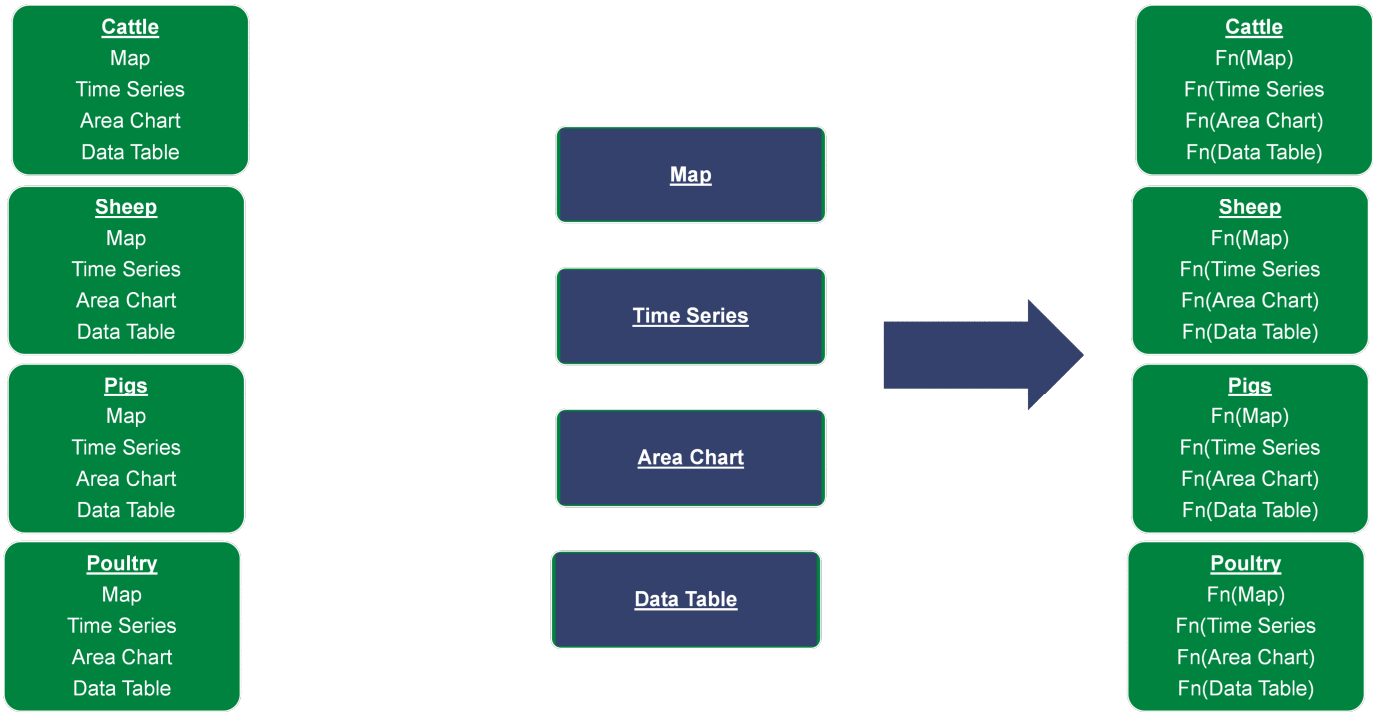
22 August 2024 10:25

The app works through 'modules' and 'functions' to keep code well organised and structured. To avoid repeating code, elements can be reused within R Shiny through functions. The example below focuses on the Livestock section of the app. Each separate page (module) contains the same general structure, but with different data / titles, etc. Instead of repeating code for the map, time series chart and area chart each time (data tables are currently independent in each module - a lot of complexity in them so standardising was difficult), these are defined within functions, which can be personalised per module with data, titles, footers, units, x/y cols. This helps keep code structured, tidy, and easy to reproduce when adding new modules.

All modules within the app need a ui and server. The ui controls more static visual elements, whilst the server works with adaptable elements, and tells the ui what to display. Likewise, any R Shiny app needs a main ui and server to operate. The structure of the app is kept very organised, so content within the core ui and server elements is limited to the essentials, relying on modules to display the content for each page.

Other key R Shiny principles include

- All elements need a unique id - consistency and uniqueness in IDs is ESSENTIAL
- Inputs = things the user can do to change visual content
- Output = the changes that the user sees as a result of inputs



# 2. Structure of App

22 August 2024 11:28

The pages within the app can generally be structured into five categories:

- **Core app functionality:**
  - The essential components any R Shiny app needs to operate

- **Modules (pages):**
  - Each module contains the content for a singular page on the app
  - Each of these needs a UI and a Server (except the home page as there is only static content)
  - Graphs/maps each require their own UI and Server, within the module's ui / server
    - These are displayed by calling functions, where the parameters of the content is defined. (Example right)
  - Content of each can be tested using 'demo' functions, displaying a mini app with only the ui and server of the page

- **Functions (graphs/maps):**
  - Where the core content of each visualisation is contained
  - These are written in a similar way to a standard more static visualisation, and then adapted to react to functions to make them reusable across different modules with different data, titles etc.
  - Most functions rely on the Highcharter library which allows interactive visualisations
    - Any R library could be used for future content, highcharter was chosen as it is a perfect tool to allow for an interactive user experience

- **Utils (utility functions):**
  - These contain elements which are reused across the app, such as footers, date inputs, and themes
  - Helpful to avoid repeating code / text
  - Allows easy updating of data, by only needing to change footers / dates in one place
  - Options util contains libraries, colour schemes and data sourcing

- **Data:**
  - These RData files are the saved output of the data.R scripts
  - These are saved to minimise data processing, so raw data does not need to be transformed every time the app is opened
  - Originate from the data.R file

- **Others:**
  - data.R = data processing file - essential for updating of new data
  - map_processing.R = converting of base geojson files to simplified format to improve efficiency (should not need re-run)
  - test.R = good place to adapt modules and test in a separate environment away from the main app
  - print_code.R = a short script which prints all the code in the entire app into a .txt file. Can be useful for debugging / finding certain code.

| Core app functionality | Modules (pages) | Functions (graphs/maps) | Utils (utility functions used across the app) | Data | Others |
|---|---|---|---|---|---|
| app.R | module_animals_summary.R | function_area_chart.R | util_options.R | census_data.RData | data.R |
| ui.R | module_beans.R | function_bar_chart.R | util_updates.R | crops_data.RData | map_processing.R |
| server.R | module_cattle.R | function_breakdown_chart.R | util_functions.R | ghg_data.RData | test.R |
| | module_cereals.R | function_data_table.R (rarely used) | hc_theme.R | module_2023.RData | print_code.R |
| | module_crops_summary.R | function_double_bar_chart.R | WWW/styles.css | total_animals.RData | |
| | module_economy_summary.R | function_line_chart,R | | | |
| | module_employees.R | function_map.R | | | |
| | module_farm_types.R | function_percentage_bar_chart.R | | | |
| | module_fertiliser_usage.R | function_summary.R | | | |
| | module_fruit.R | function_timelapse_bar_chart.R | | | |
| | module_home.R | | | | |
| | module_human_vegetables.R | | | | |
| | module_information.R | | | | |
| | module_land_use_summary.R | | | | |
| | module_legal_summary.R | | | | |
| | module_manure.R | | | | |
| | module_nitrogen_usage.R | | | | |
| | module_occupiers.R | | | | |
| | module_oilseed.R | | | | |
| | module_other_animals.R | | | | |
| | module_owned_land.R | | | | |
| | module_pigs.R | | | | |
| | module_potatoes.R | | | | |
| | module_poultry.R | | | | |
| | module_sheep.R | | | | |
| | module_soil_testing.R | | | | |
| | module_stockfeeding.R | | | | |
| | module_subsector_emissions.R | | | | |

Example of a function call within the cattle module.
This short chunk of code defines the server side content for a line chart in a simple way
This relies on function_line_chart.R and inputs the relevant parameters to display the chart.

(UI of line chart defined within the module_cattle UI
exert: tabPanel("Time Series", lineChartUI(ns("line"), note_type = 2)), )

```
# Line Chart Server
   lineChartServer(
     id = "line",
     chart_data = chart_data,
     title = "Number of cattle by category across time",
     yAxisTitle = "Number of cattle (1,000)",
     xAxisTitle = "Year",
     footer = '<div style="font-size: 16px; font-weight: bold;"><a
href="https://www.gov.scot/publications/results-scottish-agricultural-census-june-2023/documents/">
Source: Scottish Agricultural Census: June 2023</a></div>',
     x_col = "year",
     y_col = "value"
   )
```

# 3.1 How to update existing data

22 August 2024 11:50

Overview:

1. Process data in data.R
2. QA data before saving as Rdata
3. Update the utils_updates.R and ui.R files with new dates / footers
4. Run app (& likely debug!)

* If debugging, see 5. for some guidance on what may be going wrong

**<u>** Full instructions are included within data.R relating to each area **</u>**

1. **Process data in data.R & QA before saving as RData**

   Data manipulation within this section:

   - Census data
     - Further crops processing
     - Animal summary processing
   - Emissions data
   - 2023 census module data

   **General Instructions - How to update data:**
   Individual instructions are available for each section.
   Input will need to be in the same format as previous iterations unless code is edited to adapt this.
   Current data within the section primarily comes from the publicly available publication tables.
   Download the most recent tables from the publication, add them to the project directory, and modify the file path.
   Data processing is designed to be robust, but small changes within publications will likely lead to bugs needing to be fixed.
   Ensure when updating census information that all subsections are updated below.

   **Step 1: Run Libraries**
   **Step 2: Download .xlsx, insert into working directory**
   **Step 3: Change file path**
   **Step 4: QA table order / naming**
   **Step 5: Run code**

   **Census Data:**
   Xlsx file originates from the published data tables in supporting documents
   2023 version: https://www.gov.scot/publications/results-scottish-agricultural-census-june-2023/documents/

- To update, download the excel document and insert the file into the compendium working directory/file.
- Insert the file path in the file_path.

- Check table_names for changes vs. the previous year's names and order.
- If the order is incorrect, change the table number accordingly.

Try to avoid renaming tables below unless necessary as these are referenced throughout the app. If changing any names, ensure all sourcing of the data matches the changes (print_code.R can help to QA this).

The data processing was designed for the 2023 census tables, and changes to tables may cause bugs.Processing aims to standardize all tables to make them compatible with the different visualization modules.This includes removing the metadata above the tables, renaming headers, removing problematic trailing spaces, etc.

Additional processing can be included if necessary (e.g., 2022 poultry removed to emphasize change in methodology). Some data processing could be moved from within the server pages in the app to speed up loading & improve efficiency.

To update, run the code below.
QA results before saving as RData.
RData is then loaded within options.RData.

Ensure the rest of the census code (crops / fruit / veg, and animals) is run subsequently after updating the base tables.

Small changes may end up in data not being processing correctly, e.g. 2023 Table 3 is named "Table 3 " with a trailing space. This may change in future and require adapting the code.

**Emissions data:**

agri_gas = Gas Breakdown of Emissions from Scottish agriculture greenhouse gas emissions and nitrogen use
subsector_total = Agricultural emissions broken down by subsector from Scottish agriculture greenhouse gas emissions and nitrogen use
subsector_source = Breakdown of subsectors by source from Scottish agriculture greenhouse gas emissions and nitrogen use
national_total = Yearly breakdown of Scottish Emissions from Scottish Greenhouse Gas Emissions publication
As of the 2022-23 emissions publication, these were made available in an excel sheet before being inputted into R.
This could be repeated, or data could be read in from the different sources, and manipulated into the same format.
To update data, replace the file path and run the below script.
Load data from the Excel file
file_path <- "ghg_data.xlsx"

**Module 2023 data**
This data comes from the 2023 Agricultural Census Module Results
https://www.gov.scot/publications/results-from-the-scottish-agricultural-census-module-june-2023/
The script extracts tables looking at soil management, fertiliser usage, manure, nitrogen, and formats the data into formatting to be used in the modules.
This minimizes the amount of data processing done within the R Shiny app, improving running efficiency.

This script should not need to be re-run as the module data will not be updated, though it can be used
as a baseline to include future years' module data.

**IMPORTANT: When updating any data, ensure to QA it before saving / deploying to the app. The data processing has been designed to be as robust to future changes as possible, but there will probably be errors. This includes data values, column names, table ordering (likely very important).**

# 3.2 Deploying processed data

22 August 2024 12:04

Once you have updated data within data.R and saved them to their associated RData files, certain functions need updated with the new year. These can be found in util_updates.R. Be sure to update the census_year , this is used for functions across the app, alongside the footers. Ensure you include the correct link to publications, publishing dates, etc.

**Important: Ensure to update the last updated date within the create_footer function in ui.R whenever making changes to the app**

**It could be useful to use the print_code.R function to see all the app's code in one place whilst updating the yearly data. This ensures you can source any legacy year code, e.g. updating 2024 census data, search for 2023 or 2022 incase these have been missed.**

When changes to footers have been made, run the app (and probably debug! - see 5. for debugging help).

Once you have the app working, continue visual quality assurance.

Ensure the app works on a fresh R environment by clearing environment / restarting R.

Once you are certain the app is working, publish the updated app to the Scotland server (contact Jacqueline Massaya if requiring assistance).

# 4. Introducing new content

22 August 2024 12:14

**Key principles:**

- **start small**
- **use testing demo functions to test as you add new functions**
- **use reusable functions**
- **AI very helpful**
- **Integrate module within UI / server / home page**

To add a new module you need:

- Data
- UI
- Server
- Plan of what you want to include

The rest is up to you! Most modules include a sidebar for increased interactivity, but this is not essential - see soil testing.

New content is best designed with reference to existing modules so you can get used to the structure / different functions used. The use of AI was highly beneficial when designing new modules - provide it some existing modules to get a sense of structure then request what new content you want to add. Always use a testing demo function, and test independently from the main app before merging.

The following is a breakdown of some of the content in modules, and how you would add a line chart.

To get a bearing of modules, we will look at one of the most simple modules within the app:

| CODE | FUNCTIONALITY |
|---|---|
| farmTypesUI <- function(id) {<br>  ns <- NS(id)<br>  tagList(<br>  sidebarLayout(<br>  sidebarPanel(<br>  width = 3,<br>  radioButtons(ns("data_type"), "Data Type", choices = c("Holdings" = "holdings", "Area" = "area", "Total from Standard Outputs" = "total", "Average standard outputs per holding" = "average"), selected = "holdings"),<br>  conditionalPanel(<br>  condition = paste0("input['", ns("tabs"), "'] == '", ns("bar"), "'"),  # Show variable select only on Bar Chart tab<br>  uiOutput(ns("variable_select"))  # Variable select UI<br>  )<br>  ),<br>  mainPanel(<br>  id = ns("mainpanel"),<br>  width = 9,<br>  tabsetPanel(<br>  id = ns("tabs"),<br>  tabPanel("Bar Chart", barChartUI(ns("bar_chart")), value = ns("bar")),<br>  tabPanel("Data Table",<br>  DTOutput(ns("data_table")),<br>  downloadButton(ns("downloadData"), "Download Data"),<br>  generateCensusTableFooter(),<br><br>  value = ns("data"))<br>  )<br>  ) | Initial UI definition alongside unique id function<br><br>Sidebar defined<br>Width = 3 (shiny apps 12 in total)<br>Sidebar content - data type selection defined (radio buttons), then choices = the options, selected = the preset options<br><br>Main panel<br>Make up the rest of the 9<br>tabsetpanel = the submenu unique id for menu<br>tabpanel = each tab, define ui for content & value = id, tabpanel for data table, then different ui elements for dt, download button / footer, id |

```
  )
 )
}

farmTypesServer <- function(id) {
 moduleServer(id, function(input, output, session) {
  ns <- session$ns

  # Create reactive data excluding 'All' and depending only on data_type
(for chart)
  filtered_data_chart <- reactive({
   farm_type %>%
    filter(`Main farm type` != "All")
  })

  # Create reactive data for the table with commas added to numeric
values
  filtered_data_table <- reactive({
   filtered_data_chart() %>%
    mutate(across(where(is.numeric), comma))
  })

  # Get filtered data based on selected variables (only for bar chart)
  chart_data <- reactive({
   data <- filtered_data_chart()
   if (input$tabs == ns("bar") && !is.null(input$variables)) {
    data <- data %>%
     filter(`Main farm type` %in% input$variables)
   }
   data
  })

  # Select the appropriate column based on data_type
  y_col <- reactive({
   switch(input$data_type,
       "holdings" = "Holdings",
       "area" = "Hectares",
       "total" = "Total from Standard Outputs",
       "average" = "Average standard outputs per holding")
  })

  yAxisTitle <- reactive({
   switch(input$data_type,
       "holdings" = "Number of Holdings",
       "area" = "Area (hectares)",
       "total" = "Total from Standard Outputs",
       "average" = "Average Standard Outputs per Holding")
  })

  tooltip_format <- reactive({
   switch(input$data_type,
       "holdings" = "Holdings: {point.y:.0f}",
       "area" = "Area (hectares): {point.y:.2f}",
       "total" = "Total from Standard Outputs: {point.y:.0f}",
       "average" = "Average Standard Outputs per Holding: {point.y:.2f}")
  })

  # Render the data table based only on data_type selection with 20 entries
by default
  output$data_table <- renderDT({
   datatable(
    filtered_data_table() %>%
     select(`Main farm type`, y_col()),
    colnames = c("Main Farm Type", yAxisTitle()),
```

processing data - any non-reactive data processing should be done outwith the server to improve efficiency making the data reactive - this SHOULD be in the server

this allows the data to be interactive & change depending on the users inputs (variable select, etc.)

helpful to have independent data sources for graphs / data tables to keep things simple

reactive labels for when bar type changing

define data table logic - they differ from module to module that it was easier to have them independently & not as function. function probably could be done but would be a bit of a waste of time

download logic & names -

```
      options = list(pageLength = 20, scrollX = TRUE)  # Show 20 entries by
default, enable horizontal scrolling
    )
  })

  # Create a download handler for the data
  output$downloadData <- downloadHandler(
    filename = function() {
      paste("Farm_Types_", input$data_type, ".xlsx", sep = "")
    },
    content = function(file) {
      write.xlsx(filtered_data_table() %>%
                   select(`Main farm type`, y_col()), file, rowNames = FALSE)
    }
  )

  # Render the variable selection UI dynamically
  output$variable_select <- renderUI({
    choices <- unique(farm_type$`Main farm type`)
    selected <- setdiff(choices, "All")
    selectizeInput(
      ns("variables"),
      "Click within the box to select variables",
      choices = choices,
      selected = selected,
      multiple = TRUE,
      options = list(
        plugins = list('remove_button')
      )
    )
  })

  # Render the bar chart using the filtered data
  barChartServer(
    id = "bar_chart",
    chart_data = chart_data,
    title = paste("Farm types in Scotland in", census_year),
    yAxisTitle = yAxisTitle,
    xAxisTitle = "Main farm type",
    unit = input$data_type,
    footer = '<div style="font-size: 16px; font-weight: bold;"><a
href="https://www.gov.scot/publications/results-scottish-agricultural-
census-june-2023/documents/">Source: Scottish Agricultural Census: June
2023</a></div>',
    x_col = "Main farm type",
    y_col = y_col,
    tooltip_format = tooltip_format,
    maintain_order = FALSE
  )
 })
}
# Testing function for the entire module
farmTypesDemo <- function() {
 ui <- fluidPage(farmTypesUI("farm_types_demo"))
 server <- function(input, output, session) {
  farmTypesServer("farm_types_demo")
 }
 shinyApp(ui, server)
}

farmTypesDemo()
```

these need fixed!

this should be done in conditional panels, hadnt discovered them properly at this point!

bar chart server. unique id, data, title, y/x axis titles, unit (for the tooltips), footer (function from util_options), x_col, y_col ,

testing module

All new modules will follow a similar structure, defining the ui, server, and different components of each. Often these will be charts with a data table and associated downloads, so content will be very reusable across different modules.

So now we're familiar with a simple module, and want to add content - say a line chart - to the module. The good news is this should be easy as the line chart function already exists:

| | |
|---|---|
| lineChartUI <- function(id, note_type = 1) {<br>  ns <- NS(id)<br><br>  note_content <- if (note_type == 2) {<br>   "Note:<ul><br>    <li>To add a series to the chart, click inside the white box on the sidebar and select a variable.</li><br>    <li>To remove a series, click the x beside the variable name within the sidebar.</li><br>   <li>Zoom into the graph by clicking and dragging over the area you wish to focus on.</li><br>    <li>You can see data values for a specific year by hovering your mouse over the line.</li><br>   </ul>"<br>  } else {<br>   "Note:<ul><br>    <li>To add or remove a series from the chart, select/deselect the variable from the sidebar menu.</li><br>    <li>Zoom into the graph by clicking and dragging over the area you wish to focus on.</li><br>    <li>You can see data values for a specific year by hovering your mouse over the line.</li><br>   </ul>"<br>  }<br><br>  tagList(<br>   htmlOutput(ns("title")),<br>   highchartOutput(ns("line_chart")),<br>   htmlOutput(ns("footer")),<br>   div(<br>    class = "note",<br>    style = "margin-top: 20px; padding: 10px; border-top: 1px solid #ddd;",<br>    HTML(note_content)<br>   )<br>  )<br>} | Define ui of the module - function of id (for uniqueness) and note_type (differing notes if line charts use checkboxes or selectize input for variable selection) with default 1.<br><br>Text<br><br>html output - title, used across the different modules - keeps it separate and more personalisable<br>highchart - the actual line chart itself<br>footer - text below<br>html formatting |
| lineChartServer <- function(id, chart_data, title, yAxisTitle, xAxisTitle, unit = "", footer, x_col, y_col) {<br>  moduleServer(id, function(input, output, session) {<br>   ns <- session$ns<br>   reactive_colors <- reactive({ assign_colors(chart_data(), preset_colors) })<br><br>   output$title <- renderUI({<br>    year_min <- min(as.numeric(chart_data()[[x_col]]), na.rm = TRUE)<br>    year_max <- max(as.numeric(chart_data()[[x_col]]), na.rm = TRUE)<br>    HTML(paste0(" ", title, ", ", year_min, " to ", year_max, " "))<br>   })<br><br>   output$footer <- renderUI({<br>    HTML(footer)<br>   })<br><br>   output$line_chart <- renderHighchart({ | server & function options, where = " ", default = nothing<br><br>sets colour scheme<br><br>title, using the min/max years in the data<br><br>highchart line chart itself |

| | |
|---|---|
| data <- chart_data()<br>colors <- reactive_colors()<br>group_column <- setdiff(names(data), c(x_col, y_col))[1] # Assuming only one group column | data<br>colours<br>first column names |
| hc <- highchart() %>%<br>hc_chart(type = "line", zoomType = "xy") %>%<br>hc_yAxis(title = list(text = yAxisTitle)) %>%<br>hc_xAxis(title = list(text = xAxisTitle), type = "category", tickInterval = 5) %>%<br>hc_plotOptions(line = list(colorByPoint = FALSE)) %>%<br>hc_legend(align = "left", alignColumns = FALSE, layout = "horizontal") %>%<br>hc_add_theme(thm) | typical chart definition with plot options etc.<br>yAxisTitle = variable which is replaced by the function |
| unique_groups <- unique(data[[group_column]])<br>lapply(unique_groups, function(g) {<br>series_data <- data[data[[group_column]] == g, ] | |
| # Create a complete sequence of years<br>complete_years <- seq(min(series_data[[x_col]], na.rm = TRUE), max(series_data[[x_col]], na.rm = TRUE))<br>complete_series <- merge(data.frame(x = complete_years), series_data, by.x = "x", by.y = x_col, all.x = TRUE)<br>complete_series <- complete_series %>% transmute(x = as.numeric(x), y = !!sym(y_col))<br><br>hc <<- hc %>%<br>hc_add_series(name = g, data = list_parse2(complete_series), color = colors[[g]])<br>}) | think this section focuses on adding a different line for each variable - bit confusing, chatgpt to thank for it |
| hc %>%<br>hc_tooltip(<br>useHTML = TRUE,<br>headerFormat = "{point.key}<br/>",<br>pointFormatter = JS(sprintf("function() {<br>var value = this.y;<br>var formattedValue;<br>if (value >= 1000) {<br>formattedValue = value.toLocaleString(undefined, {minimumFractionDigits: 0, maximumFractionDigits: 0});<br>} else {<br>formattedValue = value.toLocaleString(undefined, {minimumFractionDigits: 0, maximumFractionDigits: 2});<br>}<br>return this.series.name + ': ' + formattedValue + ' %s';<br>}", unit))<br>)<br>})<br>})<br>} | tooltip options with custom javascript for formatting & the rounding of decimal places - another great use of AI because learning js just for this would have been impossible |

The land ownership module doesn't have any data which is compatible with a line chart as its for one year. Instead let's look at the data processing for the cattle module. To add a line chart you first need to define data, ensure it is reactive and in long format:

| | |
|---|---|
| # Processing data for Area Chart and Time Series<br>chart_data <- reactive({<br>req(input$timeseries_variables)<br>filtered_data <- number_of_cattle %>%<br>filter(`Cattle by category` %in% input $timeseries_variables) %>% | Defines that its going to be called chart_data<br>req doesn't need to be there, just a fail safe (semi useless)<br>takes number_of_cattle (from census.Rdata , variables are in Cattle by Category, pivot longer to make into 'year' and 'value' (*ENSURE NAMING WHEN PIVOTING/PROCESSING DATA IS CONSISTENT WITH |

| | |
|---|---|
| pivot_longer(cols = -`Cattle by category`, names_to = "year", values_to = "value") %>%<br>mutate(year = as.numeric(year)) # Ensure year is numeric<br>filtered_data<br>}) | OTHER MODULES). This is very important when utilising reusable modules.<br>numeric data<br>then calls data,<br>if filtering was done before could probably be simplified to<br>chart_data <- reactive({ filtered_data }) |

So now we have chart_data - we then define the line chart UI within the main panel, and the line chart server - all within the module we're looking at. For the cattle module, that looks like

```
cattleUI <- function(id) {
 ns <- NS(id)
 sidebarLayout(
  sidebarPanel(
   width = 3,
   conditionalPanel(
    condition = "input.tabsetPanel === 'Map'",
    ns = ns,
    radioButtons(
     ns("variable"),
     "Select Variable",
     choices = c(
      "Total Cattle" = "Total Cattle",
      "Total Female Dairy Cattle" = "Total Female Dairy Cattle",
      "Total Female Beef Cattle" = "Total Female Beef Cattle",
      "Total Male Cattle" = "Total Male Cattle",
      "Total Calves" = "Total Calves"
     )
    )
   ),
   conditionalPanel(
    condition = "input.tabsetPanel === 'Time Series' || input.tabsetPanel === 'Area Chart'",
    ns = ns,
    selectizeInput(
     ns("timeseries_variables"),
     "Click within the box to select variables",
     choices = unique(number_of_cattle$`Cattle by category`),
     selected = c(
      "Total Female Dairy Cattle",
      "Total Female Beef Cattle",
      "Total Male Cattle",
      "Total Calves"
     ),
     multiple = TRUE,
     options = list(
      plugins = list('remove_button')
     )
    )
   ),
   conditionalPanel(
    condition = "input.tabsetPanel === 'Data Table'",
    ns = ns,
    radioButtons(
     ns("table_data"),
     "Select Data to Display",
     choices = c("Map Data" = "map", "Time Series Data" = "timeseries"),
     selected = "map"
    )
   )
  ),
  mainPanel(
   width = 9,
```

**Sidebar**

Using selectize input, which allows you to pick multiple options from dropdown menu (useful when theres too many options to just use checkboxes)

Also uses conditional panels - these are useful when you want to define a different sidebar for each tab within a module

```
    tabsetPanel(
      id = ns("tabsetPanel"),
      tabPanel("Map", mapUI(ns("map"))),
      tabPanel("Time Series", lineChartUI(ns("line"), note_type = 2)),
      tabPanel("Area Chart", areaChartUI(ns("area"), note_type = 2)),
      tabPanel("Data Table",
               DTOutput(ns("table")),
               downloadButton(ns("downloadData"), "Download Data"),
               generateCensusTableFooter()
      )
    )
   )
  )
}


cattleServer <- function(id) {
  moduleServer(id, function(input, output, session) {
    ns <- session$ns

    # Processing data for Map
    cattle_data <- livestock_subregion %>%
      filter(`Livestock by category` %in% c(
        "Total Female Dairy Cattle",
        "Total Female Beef Cattle",
        "Total Male Cattle",
        "Total Calves",
        "Total Cattle"
      )) %>%
      select(-Scotland) %>%
      mutate(across(everything(), as.character)) %>%
      pivot_longer(cols = -`Livestock by category`, names_to = "sub_region", values_to =
"value") %>%
      mutate(value = safe_as_numeric(value))

    # Map Server
    mapServer(
      id = "map",
      data = reactive({
        req(input$variable)
        cattle_data %>% filter(`Livestock by category` == input$variable)
      }),
      footer = '<div style="font-size: 16px; font-weight: bold;"><a
href="https://www.gov.scot/publications/results-scottish-agricultural-census-
june-2023/documents/">Source: Scottish Agricultural Census: June 2023</a></div>',
      variable = reactive(input$variable),
      title = paste("Cattle distribution by region in Scotland in", census_year),
      legend_title = "Number of cattle"
    )

    # Processing data for Area Chart and Time Series
    chart_data <- reactive({
      req(input$timeseries_variables)
      filtered_data <- number_of_cattle %>%
        filter(`Cattle by category` %in% input$timeseries_variables) %>%
        pivot_longer(cols = -`Cattle by category`, names_to = "year", values_to = "value")
%>%
        mutate(year = as.numeric(year))  # Ensure year is numeric
      filtered_data
    })

    # Area Chart Server
    areaChartServer(
      id = "area",
```

**Line chart UI**
- note_type 2 as using selectizeinput above

Data manipulation

```
    chart_data = chart_data,
    title = "Number of cattle by category across time",
    yAxisTitle = "Number of cattle (1,000)",
    xAxisTitle = "Year",
    footer = '<div style="font-size: 16px; font-weight: bold;"><a
href="https://www.gov.scot/publications/results-scottish-agricultural-census-
june-2023/documents/">Source: Scottish Agricultural Census: June 2023</a></div>',
    x_col = "year",
    y_col = "value"
   )

   # Line Chart Server                                                    Server call, specifying each variable &
   lineChartServer(                                                     titles
    id = "line",
    chart_data = chart_data,
    title = "Number of cattle by category across time",
    yAxisTitle = "Number of cattle (1,000)",
    xAxisTitle = "Year",
     footer = census_footer,
    x_col = "year",
    y_col = "value"
   )

   # Data Table output
   output$table <- renderDT({
    req(input$tabsetPanel == "Data Table")
    data <- if (input$table_data == "map") {
     cattle_data %>%
      filter(`Livestock by category` == input$variable) %>%
      pivot_wider(names_from = sub_region, values_from = value) %>%
      mutate(across(where(is.numeric) & !contains("Year"), comma)) # Pivot wider for map
data
    } else {
     number_of_cattle %>%
      pivot_longer(cols = -`Cattle by category`, names_to = "year", values_to = "value")
%>%
      pivot_wider(names_from = year, values_from = value) %>%
      mutate(across(where(is.numeric) & !contains("Year"), comma)) # Pivot wider for time
series data
    }
    datatable(
     data,
     options = list(
      scrollX = TRUE,  # Enable horizontal scrolling
      pageLength = 20  # Show 20 entries by default
     )
    )
   })

   # Data Download Handler
   output$downloadData <- downloadHandler(
    filename = function() {
     if (input$table_data == "map") {
      paste("Cattle_Map_Data_", Sys.Date(), ".csv", sep = "")
     } else {
      paste("Cattle_Timeseries_Data_", Sys.Date(), ".csv", sep = "")
     }
    },
    content = function(file) {
     data <- if (input$table_data == "map") {
      cattle_data %>%
       filter(`Livestock by category` == input$variable) %>%
       pivot_wider(names_from = sub_region, values_from = value)
     } else {
```

```
        number_of_cattle %>%
          pivot_longer(cols = -`Cattle by category`, names_to = "year", values_to = "value")
%>%
          pivot_wider(names_from = year, values_from = value)
       }
       write.csv(data, file, row.names = FALSE)
     }
    )
  })
}


cattle_demo <- function() {
  ui <- fluidPage(cattleUI("cattle_test"))
  server <- function(input, output, session) {
    cattleServer("cattle_test")
  }
  shinyApp(ui, server)
}

cattle_demo()
```

# 5. Tips, tools & key small weird intricacies

22 August 2024 12:14

- **Always ensure your ids are unique and consistent** across different places they're referenced - this can be the easiest and most annoying mistake to make / fix

- **Make sure naming of variables matches other modules when adding new content** - for example, when adding a new map, the variables in the data you pass must be called sub_region (or region depending on the map) and value. These are case sensitive and the map will likely not work if they do not match.

- **Shiny/highcharter/r often does not give you error codes** - perfect example of this is the previous too points - if an id is wrong or duplicated, content wont load and R wont tell you why. often the same with maps - the map's UI will load and you'll be left looking at a blank map.

- **Aim to make code simple, tidy, and reproducible BUT sometimes its not possible / inefficient** - it's sometimes easier to not include functions for things, especially if you're not going to reuse it, or its complicated and many variables change upon different uses. Key example of this is data table / download functions - given the conditional panels that exist across the map and inconsistencies across different data tables & structuring, data inputs, labels, etc. it doesn't make sense to try and have one function to handle this for every module. You'll save a lot more time just asking AI to replicate a feature across different modules by getting it to do something once, then repeating it across modules.

- **Use of AI is highly beneficial for R**
    - Explain simply and thoroughly what you want it to do
        - It will not ask you for more details if it's not sure of something, it'll make a best guess, often leading to completely made up logic
    - Do things one step at a time
        - Don't say make me a new module like this , doing this, with this, and change this - it will not work and it'll be difficult to work out why
        - Build new content one step at a time, testing changes with testing functions each time
    - Great for simple things or complex things
        - If you don't want to code a simple thing, ask it to do it for you in plain text
        - If you don't know how to do something, it might know!
    - If you're using it regularly, the premium versions of AI are fantastic & save you so much time vs. standard versions