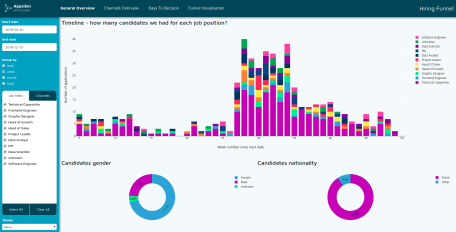
It is not always possible to create a dashboard that fully meets your expectations or requirements using only existing libraries. Maybe you want a specific function that needs to be custom built, or maybe you want to add your own style or company branding. Whatever the case, a moment might come when you need to expand and organize your code base, and dive into creating a custom solution for your project; but where to start? In this post, I will explain the relevant parts of my workflow for Shiny projects using our hiring funnel application as an example.



I will cover:

* Structuring the project folder: what goes where?
  + Managers and extracting values into settings files.
  + Using modules to organize your code.
* What does it actually take to create a custom component?

Hopefully, these topics will be as valuable to you as they have been to me!

**Structuring your project folder**

Your typical dashboard project doesn’t have a very complex structure, but this can change a lot as your project grows, so I typically try to keep things as separate as possible, and to provide some guidance for future collaborators.

I wont go over styles since these are basically an implementation of [Sass](https://github.com/rstudio/sass). Sass lets you keep your sanity while avoiding inline styling.

So what does our project folder actually look like? Here is our [hiring funnel](https://appsilon.com/journey-from-basic-prototype-to-production-ready-shiny-dashboard/) project folder for example:

│ app.R

└─── app

│ global.R

│ server.R

│ ui.R

└─── managers

│ constants\_manager.R

│ data\_manager.R

└─── settings

│ app.json

│ texts.json

└─── modules

│ header.R

│ sidebar.R

│ ui\_components.R

└─── styles

│ │ main.scss

│ ...

└─── www

│ sass.min.css

└─── assets

└─── scripts

Quite a lot to unpack, but lets go over the important bits:

**Managers and Settings**

Managers are scripts that make use of R6 classes. The constants manager has already been covered before by one of my colleagues in [the super solutions series](https://appsilon.com/super-solutions-for-shiny-architecture-3-5-softcoding-constants-in-the-app/). The data manager contains all of the abstraction when dealing with data loading and processing.

Settings has all of the values that should not be hard coded. This includes constants, texts and other values that can be extracted.

**Modules**

Modules let you easily create files for managing parts of your code. This means you can have modules for specific elements or even layout sections without bloating your main files too much.

They are great when it comes to code structure. Let’s take our header for example, instead of growing our ui.R with all of the header code, we can extract it to a separate file:

# Header.R

import("shiny")

import("modules")

export("ui")

ui\_components <- use("modules/ui\_components.R")

ui <- function(id) {

tags$header(

...

)

}

All it takes is importing the libraries you plan to use, and exporting the functions you would like to make available. You can even call other modules from inside a module!

After this we just instance the module in UI.R:

# ui.R

header <- use("modules/header.R")

And can now use by simply calling the function we want:

fluidPage(

header$ui()

...

**Custom components**

When you cannot find a component that does what you want, sometimes the only option is to create it yourself. Since we are talking about HTML components, we can expect the average component to have three main parts:  
– Layout  
– Style  
– Behavior

We have already covered modules, but how do we deal with styling and behavior? Lets take for example our navigation. What we are looking for behaves as a tab system, but where the navigation is split from the content. So we need two different ui functions:

tabs\_navigation <- function(id = "main-tabs-navigation", options) {

tagList(

tags$head(

tags$script(src = "scripts/tab-navigation.js")

),

tags$div(

id = id,

class = "tabs-navigation",

`data-tab-type`= "navigation",

lapply(options, tabs\_single\_navigation)

)

)

}

tabs\_panel <- function(

id = "main\_tab\_set",

class = "tabs-container",

tabs\_content

) {

div(

id = "main-tabs-container",

class = "tabs-container",

`data-tab-type`= "tabs-container",

lapply(tabs\_content, single\_tab),

tags$script("init\_tab\_navigation()")

)

}

By giving the different elements id’s and classes, we can use sass to easily style these components. And by including a Javascript file in the element we can load and initialize browser behavior. In this case our tab-navigation.js just initializes the first tab and binds a click event to cycle through the different tabs when clicked.

init\_tab\_navigation = function() {

$( document ).ready(function() {

$("[data-tab-type='navigation']")

.find("[data-tab-type='controller']")

.first().addClass("active")

$(`[data-tab-type="tabs-container"]`)

.find(`[data-tab-type="tab"]`)

.first().addClass("active")

$("[data-tab-type='controller']").on("click", function(e){

$(this)

.addClass("active")

.siblings(`[data-tab-type="controller"]`)

.removeClass("active")

let target = $(this).data("target")

$(`[data-tab-id="${target}"]`).addClass("active")

.siblings(`[data-tab-type="tab"]`)

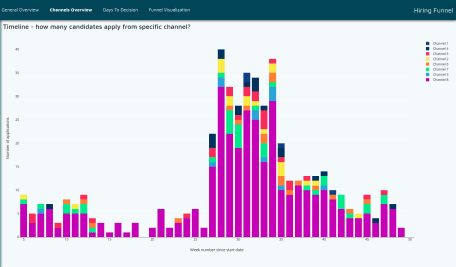
.removeClass("active")

})

});

}

It takes a bit of effort, but the result is something truly custom.



We barely scratched the surface of what can be done when it comes to custom solutions, but I hope it already gives you an idea of how to start or improve your next project!

Craving more, or have any questions? Feel free to reach out and ask!