Multi-page in {shiny}, and random thoughts about designing web applications

Something that has been bugging me for a while is the inability to really share a specific part of a `{shiny}` dashboard, at least natively, using a specific path. In other words, I've always wanted to be able to do something like my-uberapp.io/contact to share with someone a specific part of the app. And only this part.

Another need I was facing, probably born out of building web applications using NodeJS, is a **natural way to manipulate endpoints, http requests and responses**, so that I could do something pretty common in web application: a home page, and next to it a login page that verifies your identity, and redirects you to another page after setting a cookie. That of course, leads to the necessity of having the **ability to define various endpoints with a specific behavior for each**: **a unique UI, and a 18 server response**. Pure, native, multi-endpoint applications, not one big ball of UI with parts hidden using JavaScript & CSS, and a global server function that might launch computation you don't need for your page.

A few weeks back, I've decided to focus on this question. Multi-page in `{shiny}` is not new: I've found both `{shiny.router}`, and `{blaze}` that already implement a form of "multi-page".

> Note: both these packages work perfectly, and they definitely do what they are designed for. There are the product of tremendous work with smart implementations and my goal is in no way to undermine these packages. They just don't answer the need I was having, hence my new approach to this question. `{brochure}` will answer the needs for more large and complex applications built with `{shiny}`, while both the `{shiny.router}` & `{blaze}` should be easier to get started with.

As far as I saw it, these two packages didn't answer what I was looking for: "real" multi-page. Both `{shiny.router}` & `{blaze}` **still produce a form of Single Page Application**, but plays with the URL to make you feel like you're on a multi-page application.

For example, `{shiny.router}` hides the parts of the app via JS & CSS – if you browse the source code, you'll still see the hidden HTML. That also implies that they are potential conflicts between ids: you have to be sure that all your pages have unique id on their ouputs/inputs, otherwise the app doesn't render, meaning that **a "page" doesn't have its own server function**. And of course, if you look at the `Network` tab of your browser developer tool, you'll see that there is no new `GET` request made whenever a new "page" is loaded: this new page is still the same application, with parts hidden through JavaScript.

Hiding UI parts is a practice that has always been bothering me in term of performance – on a large `{shinydashboard}`, for example, **you'll be transferring the full HTML, CSS and JavaScript for the whole application when the app launches, even if the user never visits all the tabs**. `{shiny}` is very smart when it comes to transferring CSS and JS files: for example, it will only serve the external resources for the `sliderInput()` if there is one in the app. But if you've got a `{shinydashboard}` dashboard with 19 pages, and the `sliderInput()` is on page 19 and will only be seen by 0.1% of the visitors, it will still be transferred to every user, regardless of whether or not they need it. This might result in lowering the performance of the app, as it might be raising the time to 'First Meaningful Paint', and of course it is a waste of resources, especially if your application is served to people with low bandwidth, and/or browsing your app using cellular data. I know this is a question you've

probably never asked yourself before 😉 – but **performance is a pretty common question in the web development world, and something to be aware if you're serving your app at scale**. If this is something you're interested in, I suggest reading Why does speed matter? from the Google dev center, and Web Performance from Mozilla Web Docs. See also 14.2.2 Profiling {shiny} from the *Engineering Shiny* book.

And, another final thing I wanted is a way to "flush" objects when you change page: in the current implementation of `{shinydashboard}`, if you create something on `tab1`, it will still be there in `tab2`, taking some space in the RAM, even if you don't need it. It's rather convenient because you don't have to think about these things: you start a `{shiny}` session, and all the objects will still be there as long as the session lives. But that also means that as soon as the session stops, there is no way to get the values back as they only live in the RAM, inside the session.

On the other hand, with an implementation where every page gets its own `{shiny}` session, you need to find a way to identify the user with a form of `id`, save the relevant values (potentially write in a DB), then fetch these values again using the `id` stored in the browser (typically, you have a session cookie in the browser that is used as a key to search the DB). **A process forcing you to think more carefully about the data flow of your app**. What that also means is that as the `id` is a cookie in the browser, whenever the app crashes, it will be able to reload to the exact same place as long as the cookie is still in the browser (well, not exactly but you understand the idea).

So, back to our first topic – what was I looking for? My goal was to find a way to build `{shiny}` applications that are **natively** multi-page. In other words, an application that *answers* to a request on `/page2`, not an app that silently redirect to `/`, nor an application that requires playing with `/#!/page2` in the url. (Reminder, `#` is traditionally used as an anchor tag in a web page). And, of course, **I wanted each page to have its own shiny session, its own UI and server functions**.

I was also looking for a way to manipulate both the `GET` request from the server and the `httpResponse` that is sent back, for example in order to **set `httpOnly` cookies in the header of the HTTP Response or change the HTTP status code**, the same way you can manipulate these when building an application with NodeJS for example. This is something that you'd find in the `{ambiorix}` project by the amazing John Coene (who will get some SEO juice thanks to this backlink). It's an amazing approach that definitely resonated with me as I've been a NodeJS & `express.js` user for some time now, but that's still far from the way you'd build things in `{shiny}`, and will probably require a lot of coding to get an application up and running (which is good, because I love coding, but let's use our `{shiny}` knowledge for now 🙂).

To sum up, I wanted a way to get closer to how you build applications in other languages, but will as little deviation as possible from the `{shiny}` way of building apps.

Here comes `{brochure}`

As said on the top of the article, this package is still a work in progress, so you might see some changes in the near future 🙂

[Disclaimer, again] The way you will build applications with `{brochure}` is different from the way you usually build `{shiny}` apps, as we no longer operate under the single page app paradigm. Please keep this in mind and everything should be fine.

Installation

You can install the development version of `{brochure}` with:

```
remotes::install_github("ColinFay/brochure")
```

Minimal `{brochure}` App

```
library(shiny)
library(brochure)

##
## Attaching package: 'brochure'

## The following object is masked from 'package:utils':
##
##     page
```

`page()`

A `brochureApp` is a series of `page`s that are defined by an `href` (the path/endpoint where the page is available), a UI and a `server` function. This is conceptually important: **each page has its own shiny session, its own UI, and its own server**.

```
brochureApp(
  # First page
  page(
    href = "/",
    ui = fluidPage(
      h1("This is my first page"),
      plotOutput("plot")
    ),
    server = function(input, output, session){
      output$plot <- renderPlot({
        plot(iris)
      })
    }
  ),
  # Second page, without any server-side function
  page(
    href = "/page2",
    ui =  fluidPage(
      h1("This is my second page"),
      tags$p("There is no server function in this one")
    )
  )
)
```

-> In your browser, you can now navigate to `/`, and to `/page2`.

`redirect()`

Redirections can be used to redirect from one endpoint to the other:

```
brochureApp(
  page(
    href = "/",
    ui = tagList(
      h1("This is my first page")
    )
  ),
  redirect(
    from = "/nothere",
    to =  "/"
  ),
  redirect(
    from = "/colinfay",
    to =  "https://colinfay.me"
  )
)
```

-> In your browser, if you got to `/nothere`, and you'll be redirected to `/` .

`req_handlers` & `res_handlers`

Sorry what?

This is where things get more interesting.

Each page, and the global app, have a `req_handlers` and `res_handlers` parameters, that can take a **list of functions**.

An `*_handler` is a function that takes as parameter(s):

- For `req_handlers`, `req`, which is the request object (see below for when these objects are created). For example `function(req){ print(req$PATH_INFO); return(req)}`.
- For `res_handlers` , `res`, the response object, & `req` , the request object. For example `function(res, req){ print(res$content); return(res)}`.

`req_handlers` **must** return `req` & `res_handlers` **must** return `res`.

They can be used to register log, or to modify the objects, or any kind of things you can think of. If you are familiar with `express.js`, **you can think of `req_handlers` as what** `express.js` **calls "middleware"**. These functions are run when R is building the HTTP response to send to the browser (i.e, no server code has been run yet), following this process:

1. R receives a `GET` request from the browser, creating a request object, called `req`
2. The `req_handlers` are run using this `req` object, potentially modifying it
3. R creates an `httpResponse`, using this `req` and how you defined the UI
4. The `res_handlers` are run on this `httpResponse` (first app level `res_handlers`, then page level `res_handlers`), , potentially modifying it
5. The `httpResponse` is sent back to the browser

Note that if any `req_handlers` returns an `httpResponse` object, it will be returned to the browser immediately, without any further computation. This early `httpResponse` will not be

passed to the `res_handlers` of the app or the page. This process can for example be used to send custom `httpResponse`, as shown below with the `healthcheck` endpoint.

You can use formulas inside your handlers. `.x` and `..1` will be `req` for `req_handlers`, `.x` and `..1` will be `res` & `.y` and `..2` will be `req` for `res_handlers`.

Design pattern side-note: you'd probably want to define the handlers outside of the app, for better code organization (as with `log_where` below).

Example: Logging with `req_handlers()`, and building a healthcheck point

In this app, we'll log to the console every page and the time it is called, using the `log_where()` function.

```
log_where <- function(req){
  cli::cat_rule(
    sprintf(
      "%s - %s",
      Sys.time(),
      req$PATH_INFO
    )
  )
  req
}
```

For the sake of organization, we'll create functions that return pages:

```
nav_links <- tags$ul(
  tags$li(
    tags$a(href = "/", "home"),
  ),
  tags$li(
    tags$a(href = "/page2", "page2"),
  ),
  tags$li(
    tags$a(href = "/contact", "contact"),
  )
)

page_1 <- function(){
  page(
    href = "/",
    ui = function(request){
      tagList(
        h1("This is my first page"),
        nav_links,
        plotOutput("plot")
      )
    },
    server = function(input, output, session){
      output$plot <- renderPlot({
        plot(mtcars)
      })
```

```
    }
  )
}

page_2 <- function(){
  page(
    href = "/page2",
    ui =  function(request){
      tagList(
        h1("This is my second page"),
        nav_links,
        plotOutput("plot")
      )
    },
    server = function(input, output, session){
      output$plot <- renderPlot({
        plot(mtcars)
      })
    }
  )
}

page_contact <- function(){
  page(
    href = "/contact",
    ui =  tagList(
      h1("Contact us"),
      nav_links,
      tags$ul(
        tags$li("Here"),
        tags$li("There")
      )
    )
  )
}
```

We'll also build an `healthcheck` endpoint that simply returns a `httpResponse` with the 200 HTTP code.

```
# Reusing the pages from before
brochureApp(
  req_handlers = list(
    log_where
  ),
  # Pages
  page_1(),
  page_2(),
  page_contact(),
  page(
    href = "/healthcheck",
    # As this is a pure backend exchange,
    # We don't need a UI
```

```
    ui =  tagList(),
    # As this req_handler returns an httpResponse,
    # This response will be returned directly to the browser,
    # without passing through the usual dance
    req_handlers = list(
      # If you have shiny < 1.6.0, you'll need to
      # do shiny:::httpResponse (triple `:`)
      # as it is not exported until 1.6.0.
      # Otherwise, see ?shiny::httpResponse
      ~ shiny::httpResponse( 200, content = "OK")
    )
  )
)
```

If you navigate to each page, you'll see this in the console:

```
Listening on http://127.0.0.1:4879
── 2021-02-17 21:52:16 - / ─────────────────────────────
── 2021-02-17 21:52:17 - /page2 ────────────────────────
── 2021-02-17 21:52:19 - /contact ──────────────────────
```

If you go to another R session, you can check that you've got a 200 on `healthcheck`:

```
> httr::GET("http://127.0.0.1:4879/healthcheck")
Response [http://127.0.0.1:4879/healthcheck]
  Date: 2021-02-17 21:55
  Status: 200
  Content-Type: text/html; charset=UTF-8
  Size: 2 B
```

Handling cookies using `res_handlers`

`res_handlers` can be used to set cookies, by adding a `Set-Cookie` header.

Note that you can parse the cookie using `parse_cookie_string`.

```
brochure::parse_cookie_string( "a=12;session=blabla" )

##        a  session
##     "12" "blabla"
```

In the example, we'll also use `brochure::server_redirect("/")`, from the server-side to redirect the user after login.

```
# Creating a navlink
nav_links <- tags$ul(
  tags$li(
    tags$a(href = "/", "home"),
  ),
  tags$li(
    tags$a(href = "/login", "login"),
  ),
  tags$li(
    tags$a(href = "/logout", "logout"),
```

```
  )
)

home <- function(){
  page(
    href = "/",
    ui = tagList(
      h1("This is my first page"),
      tags$p("It will contain BROCHURECOOKIE depending on the last page
you've visited (/login or /logout)"),
      verbatimTextOutput("cookie"),
      nav_links
    ),
    server = function(input, output, session){
      output$cookie <- renderPrint({
        parse_cookie_string(
          session$request$HTTP_COOKIE
        )
      })
    }
  )
}

login <- function(){
  page(
    href = "/login",
    ui =  tagList(
      h1("You've just logged!"),
      verbatimTextOutput("cookie"),
      actionButton("redirect", "Redirect to the home page"),
      nav_links
    ),
    server = function(input, output, session){
      output$cookie <- renderPrint({
        parse_cookie_string(
          session$request$HTTP_COOKIE
        )
      })
      observeEvent( input$redirect , {
        # Using brochure to redirect to another page
        server_redirect("/")
      })

    },
    res_handlers = list(
      # We'll add a cookie here
      function(res, req){
        res$headers$`Set-Cookie` <- "BROCHURECOOKIE=12; HttpOnly;"
        res
      }
    )
  )
```

```
}

logout <- function(){
  page(
    href = "/logout",
    ui =  tagList(
      h1("You've logged out"),
      nav_links,
      verbatimTextOutput("cookie"),
      actionButton("redirect", "Redirect to the home page"),
    ),
    server = function(input, output, session){
      output$cookie <- renderPrint({
        parse_cookie_string(
          session$request$HTTP_COOKIE
        )
      })
      observeEvent( input$redirect , {
        # Using brochure to redirect to another page
        server_redirect("/")
      })
    },
    res_handlers = list(
      # We'll add a cookie here
      function(res, req){
        res$headers$`Set-Cookie` <- "BROCHURECOOKIE=12; Expires=Wed, 21
Oct 1950 07:28:00 GMT"
        res
      }
    )
  )
}

brochureApp(
  # Pages
  home(),
  login(),
  logout()
)
```

Random notes about design patterns

Data persistence

Every time you open a new page, a **new shiny session is launched**. This is different from what you usually do when you are building a `{shiny}` app that works as a single page application. This is no longer the case in `{brochure}`.

What that means is that **there is no data persistence in R when navigating from one page to the other**. That might seem like a downside, but I believe that it will actually be for the best: **it will make developers think more carefully about the data flow of their application, and allow a faster flush of R sessions**.

That being said, how do we keep track of a user though pages, so that if they do something in a page, it's available on another page?

To do that, you'd need to add a form of session identifier, like a cookie: this can for example be done using the `{glouton}` package if you want to manage it with JS only. You can also use the cookie example from before.

You'll also need a form of backend storage (for example with `{cachem}` here in the example, but you can also use an external DB like SQLite or MongoDB).

```
library(glouton)
# Creating a storage system
cache_system <- cachem::cache_disk(tempdir())

nav_links <- tags$ul(
  tags$li(
    tags$a(href = "/", "home"),
  ),
  tags$li(
    tags$a(href = "/page2", "page2"),
  )
)

cookie_set <- function(){
  r <- reactiveValues()

  observeEvent(TRUE, {
    # Fetch the cookies using {glouton}
    r$cook <- fetch_cookies()

    # If there is no stored cookie for {brochure}, we generate it
    if (is.null(r$cook$brochure_cookie)){
      # Generate a random id
      session_id <- digest::sha1(paste(Sys.time(), sample(letters,
16)))
      # Add this id as a cookie
      add_cookie("brochure_cookie", session_id)
      # Store in in the reactiveValues list
      r$cook$brochure_cookie <- session_id
    }
    # For debugging purpose
    print(r$cook$brochure_cookie )
  }, once = TRUE)
  return(r)
}

page_1 <- function(){
  page(
    href = "/",
    ui = tagList(
      h1("This is my first page"),
      nav_links,
```

```
      # The text enter on page 1 will be available on page 2, using
      # a session cookie and a storage system
      textInput("textenter", "Enter a text"),
      actionButton("save", "Save my text and go to page2")
    ),
    server = function(input, output, session){
      r <- cookie_set()
      observeEvent( input$save , {
        # Use the session id to save on the cache system
        cache_system$set(
          paste0(
            r$cook$brochure_cookie,
            "text"
          ),
          input$textenter
        )
        server_redirect("/page2")
      })
    }
  )
}

page_2 <- function(){
  page(
    href = "/page2",
    ui =  tagList(
      h1("This is my second page"),
      nav_links,
      # The text enter on page 1 will be available here, reading
      # the storage system
      verbatimTextOutput("textdisplay")
    ),
    server = function(input, output, session){
      r <- cookie_set()
      output$textdisplay <- renderPrint({
        # Getting the content value based on the session cookie
        cache_system$get(
          paste0(
            r$cook$brochure_cookie,
            "text"
          )
        )
      })
    }
  )
}

brochureApp(
  # Setting {glouton} globally
  use_glouton(),
  # Pages
  page_1(),
```

```
  page_2()
  # Redirections
)
```

Working with golem

`{golem}` is a small project that is starting to get some traction in the `{shiny}` world. I'm still not sure of its relevance, but I've heard that some of you are using it for some reasons, so I suppose you might want to make `{brochure}` work inside a `{golem}` based app.

To adapt your `{golem}` based application to `{brochure}`, here are the two steps to follow:

- Remove the app_server.R file, and the top of app_ui => You'll still need `golem_add_external_resources()`.
- Build the pages inside separate R scripts, following the example from this `README`.

```
.
├── DESCRIPTION
├── NAMESPACE
├── R
│   ├── app_config.R
│   ├── home.R ### YOUR PAGE
│   ├── login.R ### YOUR PAGE
│   ├── logout.R ### YOUR PAGE
│   └── run_app.R ### YOUR PAGE
├── dev
│   ├── 01_start.R
│   ├── 02_dev.R
│   ├── 03_deploy.R
│   └── run_dev.R
├── inst
│   ├── app
│   │   └── www
│   │       ├── favicon.ico
│   └── golem-config.yml
├── man
│   └── run_app.Rd
```

- Replace `shinyApp` with `brochureApp` in `run_app()`, add the external resources, then your pages.

```
run_app <- function(
  onStart = NULL,
  options = list(),
  enableBookmarking = NULL,
  ...
) {
  with_golem_options(
    app = brochureApp(
      # Putting the resources here
      golem_add_external_resources(),
      home(),
```

```
      login(),
      logout(),
      onStart = onStart,
      options = options,
      enableBookmarking = enableBookmarking
    ),
    golem_opts = list(...)
  )
}
```

Improvement & further work

- Testers and apps. If you want to give `{brochure}` a shot, please do! It’s still a young project that needs more testing, so if you feel like you want to build an app with it, please do. …