**Background**

Shiny apps are a great way to share information and empower your users. Sometimes you want to make sure that only authenticated and authorized users will be able to view your shiny apps.

There are a number of ways to make sure only certain users have access to your apps. For example, you can subscribe to the professional plan in [shinyapps.io](http://shinyapps.io) which has this option built-in. You can program the authentication flow internally by yourself, or you just use a 3rd party service such as google firebase, AWS Cognito, Auth0, or others).

The benefit of using a dedicated service is that you get a lot of features which will be a serious headake to program yourself, such as social logins, two factor authentication, logs, and user blocks on suspicious attempts (or warnings on unauthorized attempts, depending on settings).

The down side is that it takes some time to implement. In this guide I aim to make the process as simple and painless as possible, using the Amazon Web Service’s authentication solution, called *AWS Cognito*. But first, some theory about authentication.

**How authentication works**

The logic behind authentication with AWS Cognito (or similar alternatives) is that you direct your users to a login page hosted by AWS, in which the user completes a process which confirms the user’s indentity. For example, by entering an e-mail and password, or by using a social sign-in (i.e., login via gmail, amazon, facebook). Then, once Cognito is finished, the user is redirected to your app with a URL variable which contain a specially issued code (i.e., <https://your-app-address/?code=AMAZON_ISSUED_CODE>.

Then, you use an http request (i.e., with package httr) to query the Cognito API with this code, and in return you receive the information behind this code (i.e., the user’s *token*, with information such as the name of the user, what is the user’s email, validity of the token, etc.). This httr query is performed by using a password known only to you (i.e., only your app “knows” this password, this is not the user’s password).

The code is usable only once, and the token is valid for a limited duration, to minimize the risk that an unauthorized party will hijack the token and re-use it to access your app.

After authenticating the user, you can authorize the user according to privileges (which you would have to manage within your app, i.e. if the users email is X, then he can view Y).

This process description was a very simplified, down-to-earth, nutshell description of oauth2. It might be inaccurate, but it will be enough for our goal here which is to actually implement it within a shiny app, integrating to Cognito. If you wish, you can find more information about oauth2 in detail [here](https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2).

Let’s get to business.

**Step 1: Define a user pool**

This step is actually performed within the AWS Console. Log into your AWS console and find the Cognito service.  
Click on “Manage User Pools”, and then create a new user pool. The step-by-step wizard is pretty self explanatory, so I’ll focus on the important things:

* Make sure that you require a relevant field upon your user sign-up which you can “count on” in order to perform user authorization within your app based on that field later on. I usually check the email address as a required field, and then add logic in my app which maps email addresses to what each user is allowed to view.
* Multi-factor authentication can be “off”, “optional” or “required”. If your app contains sensitive information, then you should consider making it required.

**Important!** in the step where you are asked “Which app clients will have access to this user pool?” click on “Add an app client”. Give your app a name, the deafult options are sufficient so you shouldn’t change anything.

Make sure you click on “Show Details” after you added your app and document the App client id and the App client secret. You will need them later on to interact with the Cognito API.

**Optional**: right after you add your app and click “next step”, you will have a chance to add functions triggered by the various steps of the authentication flow. If you know what AWS Lambda functions are (and you defined such functions in your account) you can choose to trigger them depending on the authentication flow.

Complete the wizard and create your user pool.

**Email communications**

You must have the AWS SES (simple email service) configured properly, in order for the registration of new users and “forgot password” flows to work. By default, SES is in sandbox mode, which means you can only register users with pre-verified emails. Defining SES is outside the scope of this guide, but note that you have to open a ticket in the AWS support center, asking for these privilleges.

Make sure you supply AWS support with a lot of information about how you make sure emails don’t bounce, and about spam prevention. Even though it’s trivial, since this is an internal AWS system using the email service, they made me jump through hoops, untill granting me a 50k daily email cap, which is more than enough for me.

**App client settings**

Under the “App integration -> App client settings” you need to add the Callback URL of your app (where the user is directed upon login). For example, if your app is going to be hosted on [shinyapps.io](http://shinyapps.io) that would be: <https://YOUR_USER_NAME.shinyapps.io/YOUR_APP_NAME>. Your sign-out url can be the same, if you want the app to allow the user to restart the login, or a different page showing that the user has logged out.

Under OAuth 2.0/“Allowed OAuth Flows” you should check the: *Authorization code grant*. This is the authentication flow we are going to use for our shiny app. The “implicit grant” is not as secure, and the “client credentials” is used for machine-to-machine authentication.

Under “Allowed OAuth Scopes” check the options by which you are going to recognize your users within the shiny app’s logic. I.e., if you are going to show specific data by the user’s email address than make sure you check the “email” under allowed OAuth scopes.

Set a domain name for your login screen and customize the UI of the login screen if you wish.

You can see in the following screenshot, that I’m using this authentication with one of my apps hosted in a [shinyapps.io](http://shinyapps.io) domain, under my account.

**Enable identity provides (Optional)**

If you want to offer your users a social login such as Facebook or Google, you would need to issue the proper credentials via google console and facebook. This is a nice addition, but is out of the scope of this guide.

Finally, we get into the R code part of this post.

**Step 2: Authorization code (within R)**

Now we need to add logic to our shiny app which will redirect the user to the AWS Cognito login page, and once the user authenticates and redirected to the shiny app, our shiny app will verify the token’s validity.

Very basically, the Shiny app should read query url variables, and:

1. If no variables appear, show a login button to the user (which will redirect to the AWS Cognito login screen with the proper parameters).
2. If a url variable called code appears, our app will read its value, and use AWS Cognito to apply a second layer of verification and identification according to the code (read the token issued by Cognito).
3. If the user is logged on, show a “logout” button which will redirect the user into AWS Cognito logout link.

**Redirect links to login/logout screen (AUTHORIZATION, LOGOUT Endpoints)**

There are two “endpoints” (urls) that your users will be redirected to:

1. During their login flow into the app, or;
2. After they click “logout”.

The first is the “AUTHORIZATION Endpoint”. It is a redirection of the client to a url of the following form (I have already put it into a paste0 command, which we will later use in our app).

base\_cognito\_url <- "<https://YOUR_PREEDEFINED_AWS_COGNITO_DOMAIN.amazoncognito.com/>"

app\_client\_id <- "YOUR\_APP\_CLIENT\_ID"

app\_client\_secret <- "YOUR\_APP\_CLIENT\_SECRET"

redirect\_uri <- "<https://YOUR_APP/redirect_uri>"

# e.g., if you are using [shinyapps.io](http://shinyapps.io) this would be:

# <https://ACCOUNT_NAME.shinyapps.io/YOUR_APP_NAME>

aws\_auth\_redirect <-

paste0(

base\_cognito\_url,

"oauth2/authorize?",

"response\_type=code&",

"client\_id=", app\_client\_id, "&",

"redirect\_uri=", redirect\_uri, "&",

"state=appredirect"

)

aws\_auth\_redirect

## [1] "<https://YOUR_PREEDEFINED_AWS_COGNITO_DOMAIN.amazoncognito.com/oauth2/authorize?response_type=code&client_id=YOUR_APP_CLIENT_ID&redirect_uri=https://YOUR_APP/redirect_uri&state=appredirect>"

You can also specify the “scopes” (what information should Cognito hold for your next query, email, phone, etc.). If you don’t specify any scopes, all the information available on the user will be provided (see the next section of this post “**Querying Cognito with the grant code**”).

The second endpoint is the “LOGOUT Endpoint” which will logout the user. It is important to provide a logout button so that users can safely close your app, without worrying about other users in the same computer abusing their credentials.

aws\_auth\_logout <-

paste0(

base\_cognito\_url, "logout?",

"client\_id=", app\_client\_id, "&",

"logout\_uri=", redirect\_uri

)

aws\_auth\_logout

## [1] "<https://YOUR_PREEDEFINED_AWS_COGNITO_DOMAIN.amazoncognito.com/logout?client_id=YOUR_APP_CLIENT_ID&logout_uri=https://YOUR_APP/redirect_uri>"

Once a user has completed the login process (via the authorization endpoint), he will be redirected to your app (the link you provided in the redirect\_uri and in the Cognito setup at step 1). If the login is successful, the user will return with a url variable called code, i.e. [https://YOUR\_APP/redirect\_uri?code=####-####-####-####](https://YOUR_APP/redirect_uri?code=#%23%23%23-%23%23%23%23-%23%23%23%23-%23%23%23%23). The next step will be to make sure that this code is indeed valid, and to check who is the user trying to access behind this code.

**Querying Cognito with the grant code**

This is a crucial part, in which we make sure that the user is indeed valid, and allowed to access your app. We’re going to use the httr package for that.

Let’s assume we have already pulled the authorization code from the Shiny app’s url variables (we’re going to show how to do that in step 3).

We’re going to build a function which gets the code as an argument and provides the user’s information (or an error if the user is not authenticated or there was a different failure). I usually place this code in my global.r file, which is a part of the shiny app’s bundle (ui.r, server.r, global.r), and is used to define an environment variables and functions which will be availble to the shiny app. You can also place it at the begining of the server.r if you don’t want a global.r file. If you are using a single app.r just put it before the app itself.

Here is the code that goes into your global.r file:

base\_cognito\_url <- "<https://YOUR_DOMAIN.YOUR_AMAZON_REGION.amazoncognito.com/>"

app\_client\_id <- "YOUR\_APP\_CLIENT\_ID"

app\_client\_secret <- "YOUR\_APP\_CLIENT\_SECRET"

redirect\_uri <- "<https://YOUR_APP/redirect_uri>"

library(httr)

app <- oauth\_app(appname = "my\_shiny\_app",

key = app\_client\_id,

secret = app\_client\_secret,

redirect\_uri = redirect\_uri)

cognito <- oauth\_endpoint(authorize = "authorize",

access = "token",

base\_url = paste0(base\_cognito\_url, "oauth2"))

retrieve\_user\_data <- function(user\_code){

failed\_token <- FALSE

# get the token

tryCatch({token\_res <- oauth2.0\_access\_token(endpoint = cognito,

app = app,

code = user\_code,

user\_params = list(client\_id = app\_client\_id,

grant\_type = "authorization\_code"),

use\_basic\_auth = TRUE)},

error = function(e){failed\_token <<- TRUE})

# check result status, make sure token is valid and that the process did not fail

if (failed\_token) {

return(NULL)

}

# The token did not fail, go ahead and use the token to retrieve user information

user\_information <- GET(url = paste0(base\_cognito\_url, "oauth2/userInfo"),

add\_headers(Authorization = paste("Bearer", token\_res$access\_token)))

return(content(user\_information))

}

**Step 3: define your Shiny app’s server.r and ui.r**

In our shiny app, we need to pull the code and use the retrieve\_user\_data function we’ve just defined as part of our verification of the user. Here is the code we will use for this. This should go into the server.r file.

library(shiny)

library(shinyjs)

# define a tibble of allwed users (this can also be read from a local file or from a database)

allowed\_users <- tibble(

user\_email = c("[user1@example.com](mailto:user1@example.com)",

"[user2@example.com](mailto:user2@example.com)"))

function(input, output, session){

# initialize authenticated reactive values ----

# In addition to these three (auth, name, email)

# you can add additional reactive values here, if you want them to be based on the user which logged on, e.g. privileges.

user <- reactiveValues(auth = FALSE, # is the user authenticated or not

name = NULL, # user's name as stored and returned by cognito

email = NULL) # user's email as stored and returned by cognito

# get the url variables ----

observe({

query <- parseQueryString(session$clientData$url\_search)

if (!("code" %in% names(query))){

# no code in the url variables means the user hasn't logged in yet

showElement("login")

} else {

current\_user <- retrieve\_user\_data(query$code)

# if an error occurred during login

if (is.null(current\_user)){

hideElement("login")

showElement("login\_error\_aws\_flow")

showElement("submit\_sign\_out\_div")

user$auth <- FALSE

} else {

# check if user is in allowed user list

# for more robustness, use stringr::str\_to\_lower to avoid case sensitivity

# i.e., (str\_to\_lower(current\_user$email) %in% str\_to\_lower(allowed\_users$user\_email))

if (current\_user$email %in% allowed\_users$user\_email){

hideElement("login")

showElement("login\_confirmed")

showElement("submit\_sign\_out\_div")

user$auth <- TRUE

user$email <- current\_user$email

user$name <- current\_user$name

# ==== User is valid, continue prep ====

# show the welcome box with user name

output$confirmed\_login\_name <-

renderText({

paste0("Hi there!, ",

user$name)

})

# ==== Put additional login dependent steps here (e.g. db read from source) ====

# ADD HERE YOUR REQUIRED LOGIC

# I personally like to select the first tab for the user to see, i.e.:

showTab("main\_navigation", "content\_tab\_id", select = TRUE)

# (see the next chunk for how this tab is defined in terms of ui elements)

# ==== Finish loading and go to tab ====

} else {

# user not allowed. Only show sign-out, perhaps also show a login error message.

hideElement("login")

showElement("login\_error\_user")

showElement("submit\_sign\_out\_div")

}

}

}

})

# This is where you will put your actual elements (the server side that is) ----

# For example:

output$some\_plot <- renderPlot({

# \*\*\* THIS IS EXTREMELY IMPORTANT!!! \*\*\*

validate(need(user$auth, "No privileges to watch data. Please contact support."))

# since shinyjs is not safe for hiding content, make sure that any information is covered

# by the validate(...) expression as was specified.

# Rendered elements which were not preceded by a validate expression can be viewed in the html code (even if you use hideElement).

# only if user is confirmed the information will render (a plot in this case)

plot(cars)

})

}

The accompanying user interface (user.r) will look like the following:

library(shiny)

library(shinyjs)

fluidPage(

useShinyjs(), # to enable the show/hide of elements such as login and buttons

hidden( # this is how the logout button will like:

div(

id = "submit\_sign\_out\_div",

a(id = "submit\_sign\_out",

"logout",

href = aws\_auth\_logout,

style = "color: black;

-webkit-appearance: button;

-moz-appearance: button;

appearance: button;

text-decoration: none;

background:#ff9999;

position: absolute;

top: 0px; left: 20px;

z-index: 10000;

padding: 5px 10px 5px 10px;"

)

)

),

navbarPage(

"Cognito auth example",

id = "main\_navigation",

tabPanel(

"identification",

value = "login\_tab\_id",

h1("Login"),

div(

id = "login",

p("To login you must identify with a username and password"),

# This defines a login button which upon click will redirect to the AWS Cognito login page

a(id = "login\_link",

"Click here to login",

href = aws\_auth\_redirect,

style = "color: black;

-webkit-appearance: button;

-moz-appearance: button;

appearance: button;

text-decoration: none;

background:#95c5ff;

padding: 5px 10px 5px 10px;")

),

hidden(div(

id = "login\_error\_aws\_flow",

p("An error has occurred."),

p("Please contact support")

)),

hidden(

div(

id = "login\_confirmed",

h3("User confirmed"),

fluidRow(

textOutput("confirmed\_login\_name")),

fluidRow(

p("Use the menu bar to navigate."),

p(

"Don't forget to logout when you want to close the system."

)

)

)

),

),

tabPanel("Your actual content",

value = "content\_tab\_id",

fluidRow(plotOutput("some\_plot")))

)

)

**Conclusions**

The post contains essential things you need in order to get started with AWS Cognito authentication for your shiny apps.

You can extend this process to any authentication service (for example, digital ocean has a similar service to Cognito). There are some packages which implement the entire process for other services, like googleAuthR for a gmail login (see this link)[[https://code.markedmondson.me/googleAuthR/articles/google-authentication-types.html](https://code.markedmondson.me/googleAuthR/articles/google-authentication-types.html" \t "_blank)],

If you found this post useful, let me know!, either in comments below, or twitter, or email.

As always, be careful of how you implement this process in your own apps, to make sure there are no security risks or loopholes. Also, DISCLAIMER: The information in this post is free, you can use this however like. Note that it is published with the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.