login: User Authentication for Shiny Applications

ShinyConf 2024

Jason Bryer, Ph.D.

CUNY SPS Data Science and Information Systems

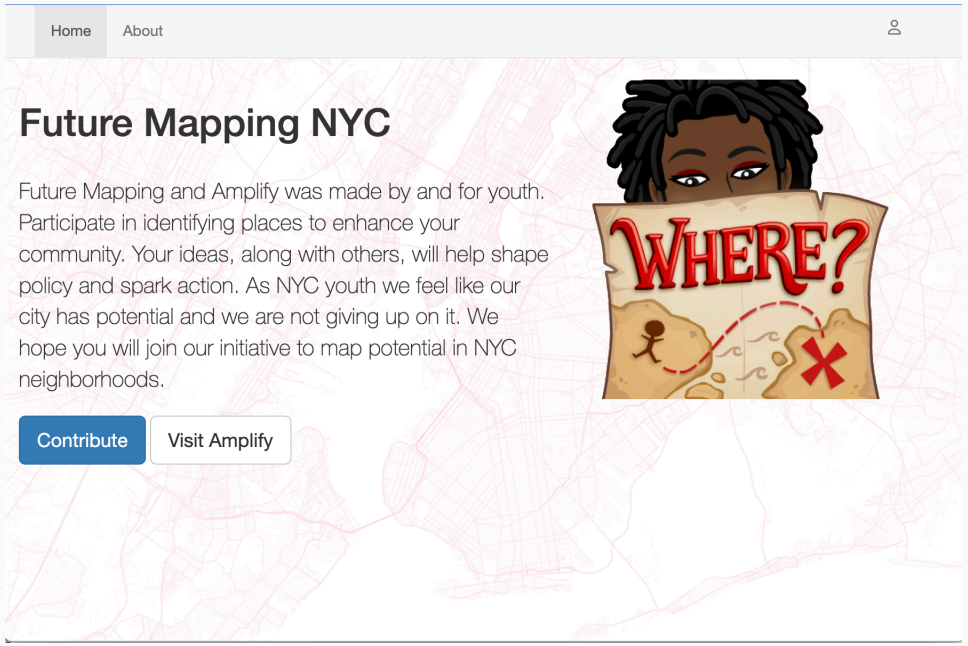
April 18, 2024

Shiny authentication is not new...

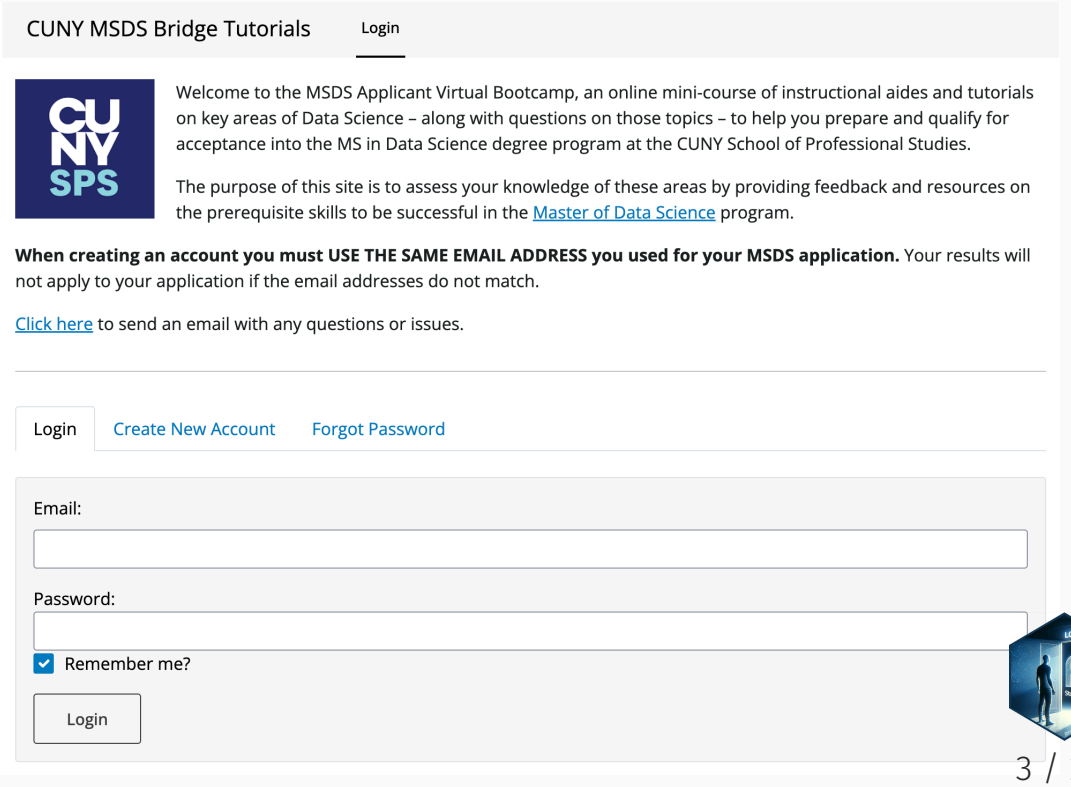
- `shinymanager` - This is a great solution if you want to limit access to your entire Shiny application. This is accomplished by passing your UI (e.g. `fluidPage` object) to the `shinymanager::secure_app` function.
- `shinyauthr` - This is another nice solution if you have a pre-existing database of users.
- Posit Connect - This is a paid product and allows you to protect your entire Shiny application outside of your application code.
- Standard and Professional plans on shinyapps.io - Similar to Posit Connect, this is a paid hosting service.

Motivation for creating another this package

FutureMapping.org is a project where young people in New York City can identify locations in their community that are assets, problems, or something else.



Master of Data Science (MSDS) at CUNY School of Professional Studies Bridge Tutorials



How is `login` different?

The `login` package provides some important features missing in the other Shiny authentication frameworks, namely:

- Users can create their own accounts. Optionally, you can require users to validate their email before creating the account.
- Users can reset their passwords via email.
- Allow user credentials to be stored in any `DBI` compatible database.
- Remember users by setting a cookie (this is optional both from the server and user point-of-view).
- Will log user activity for logging in and out.

Security

Disclaimer: I am not security researcher, but have attempted to implement the best practices to the best of my knowledge. If you have any suggestions feel free to open a Github issue.

Here are some of the security features implemented by the `login` package:

- Passwords are encrypted client side (in the browser) using a MD5 hash. There are a few advantages of doing this client side:
    - The password is never sent over the network in the clear. *You should still deploy your Shiny applications over https*. Posit has instructions to do this.
    - The Shiny server never sees the unencrypted password.
- Salt - Salting is the processes of encrypting the password a second time before storing the password in the database. To enable set the `salt` parameter of `login::login_server()`. Additionally, you can specify the encryption algorithm to use (defaults to `sha512` but any algorithm supported by `digest::digest()` is valid).

Getting started

You can install the package from Github using the `remotes` package (should be on CRAN soon):

```
remotes::install_github('jbryer/login')
```

There are three example Shiny applications included in the package:

- https://github.com/jbryer/login/blob/main/inst/login_demo_simple/app.R
- https://github.com/jbryer/login/blob/main/inst/login_template/app.R
- https://github.com/jbryer/login/blob/main/inst/login_modal/app.R

I recommend putting this at the top of your `app.R` or `global.R` file:

```
library(login)
APP_ID <- 'my_login_app'
```

Server Side

```
  USER <- login::login_server(
   id = APP_ID,
  db_conn = RSQLite::dbConnect(RSQLite::SQLite(),
    'users.sqlite'),
  users_table = "users",
  emailer = emayili_emailer(
    email_host = EMAIL_HOST,
    email_port = EMAIL_PORT,
    email_username = EMAIL_USERNAME,
    email_password = EMAIL_PASSWORD,
    from_email = RESET_PASSWORD_FROM_ADDRESS ),
  additional_fields = c('first_name' = 'First Name',
                        'last_name' = 'Last Name'),
  cookie_name = "loginusername",
  cookie_expiration = 30,
  salt = 'mysupersecretsalt'
)
```

The `login` package is implemented as a Shiny module. Within your server function you must call `login::login_server()`. The code on the left includes some of the basic features. The function will return a `reactiveValues` object with the following elements:

- `logged_in` - This is TRUE or FALSE.
- `username` - The username if they have successfully logged in (i.e. their email address).

Defining database connection

```
USER <- login::login_server(
  id = APP_ID,
   db_conn = RSQLite::dbConnect(RSQLite::SQLite(),
     'users.sqlite'),
   users_table = "users",
  emailer = emayili_emailer(
    email_host = EMAIL_HOST,
    email_port = EMAIL_PORT,
    email_username = EMAIL_USERNAME,
    email_password = EMAIL_PASSWORD,
    from_email = RESET_PASSWORD_FROM_ADDRESS ),
  additional_fields = c('first_name' = 'First Name',
                        'last_name' = 'Last Name'),
  cookie_name = "loginusername",
  cookie_expiration = 30,
  salt = 'mysupersecretsalt'
)
```

All the `login` credentials are stored using the `DBI` interface. `DBI` currently supports RSQLite, RMariaDB, odbc, and bigrquery natively, but there is an extensive list of other bakends available.

The `users_table` parameter defines the table name in the database to store credentials. If the table doesn't exist, it will be created upon first run.

Email

```
USER <- login::login_server(
  id = APP_ID,
  db_conn = RSQLite::dbConnect(RSQLite::SQLite(),
    'users.sqlite'),
  users_table = "users",
   emailer = emayili_emailer(
     email_host = EMAIL_HOST,
     email_port = EMAIL_PORT,
     email_username = EMAIL_USERNAME,
     email_password = EMAIL_PASSWORD,
     from_email = RESET_PASSWORD_FROM_ADDRESS ),
  additional_fields = c('first_name' = 'First Name',
                        'last_name' = 'Last Name'),
  cookie_name = "loginusername",
  cookie_expiration = 30,
  salt = 'mysupersecretsalt'
)
```

An important feature of the `login` package is the ability to have users verify their email address when creating an account and for them to reset their passwords. The `emailer` parameter must define a function with three parameters: `to_email`, `subject`, and `message`. The package includes the `emayili_emailer()` function that uses the `emayili` package to send emails using SMTP. If you wish to implement your own emailer, use the following template:

```
my_emailer <- function(to_email, subject, message) {
    # Implementation here
}
```

Additional user information

```
USER <- login::login_server(
  id = APP_ID,
  db_conn = RSQLite::dbConnect(RSQLite::SQLite(),
    'users.sqlite'),
  users_table = "users",
  emailer = emayili_emailer(
    email_host = EMAIL_HOST,
    email_port = EMAIL_PORT,
    email_username = EMAIL_USERNAME,
    email_password = EMAIL_PASSWORD,
    from_email = RESET_PASSWORD_FROM_ADDRESS ),
   additional_fields = c('first_name' = 'First Name',
                         'last_name' = 'Last Name'),
  cookie_name = "loginusername",
  cookie_expiration = 30,
  salt = 'mysupersecretsalt'
)
```

At a minimum, the `login` package will capture the users' email address and (encrypted) password. If you wish to capture more information when the user creates an account, you can specify the desired fields with the `additional_fields` parameter. This is a character vector where the names are the database field names and the values are the labels users will see when creating an account. Also, the values will be available in the returned `reactiveValues`, that is `USER$first_name` and `USER$last_name` will be set.

Cookies

```
USER <- login::login_server(
  id = APP_ID,
  db_conn = RSQLite::dbConnect(RSQLite::SQLite(),
    'users.sqlite'),
  users_table = "users",
  emailer = emayili_emailer(
    email_host = EMAIL_HOST,
    email_port = EMAIL_PORT,
    email_username = EMAIL_USERNAME,
    email_password = EMAIL_PASSWORD,
    from_email = RESET_PASSWORD_FROM_ADDRESS ),
  additional_fields = c('first_name' = 'First Name',
                        'last_name' = 'Last Name'),
   cookie_name = "loginusername",
   cookie_expiration = 30,
  salt = 'mysupersecretsalt'
)
```

If you wish to allow users login state to be saved between sessions, set the `cookie_name` parameter. This will be the name of the cookie stored in their browser. The `cookie_exppiration` is how long the cookie is valid in days. Set this parameter to `NULL` to disable cookies all together.

Note that if you have cookies enabled, users can still opt-out by unchecking the "remember me" check box.

Salting

```
USER <- login::login_server(
  id = APP_ID,
  db_conn = RSQLite::dbConnect(RSQLite::SQLite(),
    'users.sqlite'),
  users_table = "users",
  emailer = emayili_emailer(
    email_host = EMAIL_HOST,
    email_port = EMAIL_PORT,
    email_username = EMAIL_USERNAME,
    email_password = EMAIL_PASSWORD,
    from_email = RESET_PASSWORD_FROM_ADDRESS ),
  additional_fields = c('first_name' = 'First Name',
                        'last_name' = 'Last Name'),
  cookie_name = "loginusername",
  cookie_expiration = 30,
   salt = 'mysupersecretsalt'
)
```

To enable salting, set the `salt` parameter. Additionally, you can use a different algorithm (sha512 is the default) using the `salt_algo` parameter. This parameter is passed to the `digest::digest()` function.

There are even more options...

- `activity_table` the name of the table in the database to log login and logout activity.
- `reset_password_subject` the subject of password reset emails.
- `new_account_subject` the subject used for verifying new accounts.
- `verify_email` if true new accounts will need to verify their email address before the account is crated. This is done by sending a six digit code to the email address.
- `username_label` label used for text inputs of username.
- `password_label` label used for text inputs of password.
- `create_account_label` label for the create account button.
- `create_account_message` Email message sent to confirm email when creating a new account. Include `\%s` somewhere in the message to include the code.
- `reset_email_message` Email message sent to reset password. Include `\%s` somewhere in the message to include the code.
- `enclosing_panel` the Shiny element that contains all the UI elements. The default is `shiny::wellPanel()`. If you wish a more subtle appearance `htmltools::div()` is a reasonable choice.
- `code_length` the number of digits of codes emailed for creating accounts (if `verify_email == TRUE`) or resetting passwords.
- `shinybusy_position` Position of the spinner when sending emails. See `shinybusy::use_busy_spinner()` for more information.
- `shinybusy_spin` Style of the spinner when sending emails. See `shinybusy::use_busy_spinner()` for more information.

Using the authentication state in the Shiny server

The `login::login_server()` returns a `reactiveValues` object. Therefore you can use the `USER$logged_in` within your Shiny server to change the behavior of the application based upon the users' login state. Since this is reactive, if the user logs out, the code will be re-executed.

```
if(USER$logged_in) {
    username <- USER$username
    # The user is logged in, do something.
} else {
    # The user is not logged in.
}
```

Shiny UI: Logging in and out

There are a number of functions included to provide the user interface elements for logging in, logging out, resetting password, and creating an account. The `login_ui()` function provides an interface for the user to login. Similarly, the `logout_button()` provides a button for the user to logout. These interface elements will only display if the user is logged out or logged in, respectively.

```
login::login_ui(id = APP_ID)
```

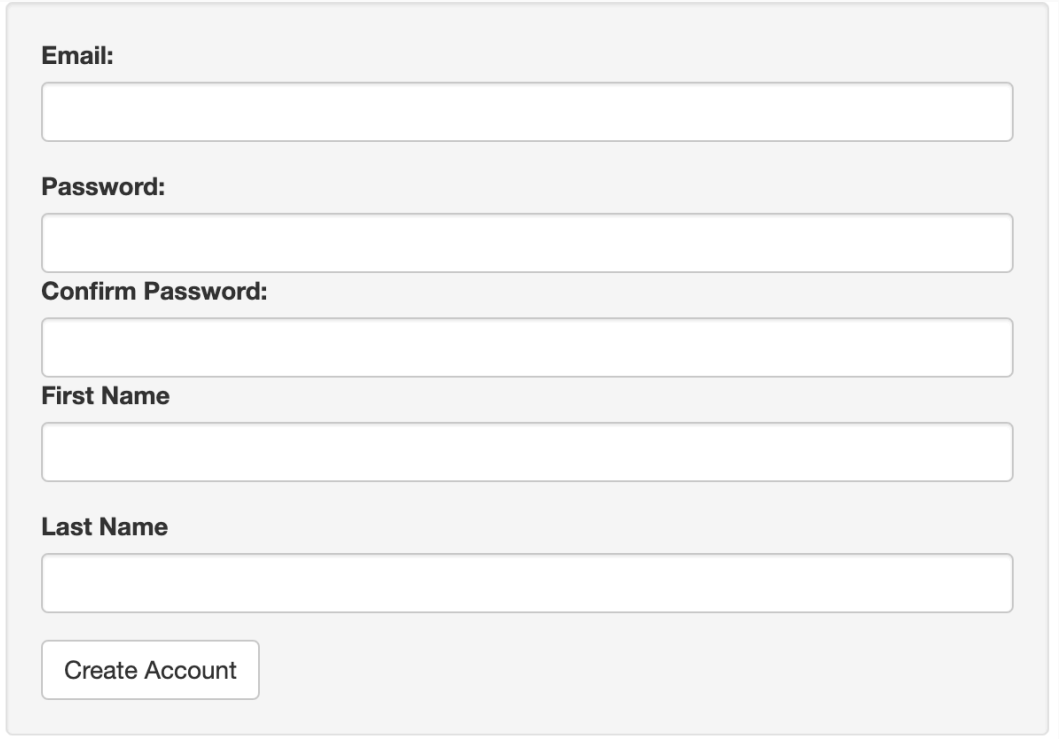

```
logout_button(
  id = APP_ID,
  label = "Logout",
  icon = shiny::icon("right-from-bracket"))
```

Shiny UI: Creating an account

The `new_user_ui()` function will provide the fields necessary to create a new account. Note that there are additional text fields for the `additional_fields` parameter specified in the `login::login_server()` function earlier.

If the `verify_email = TRUE` in the `login::login_server()` then the user will be required to verify their email address before their account is created (and saved to the database). The interface is the same as steps 2 and 3 of the password reset process described on the next slide.

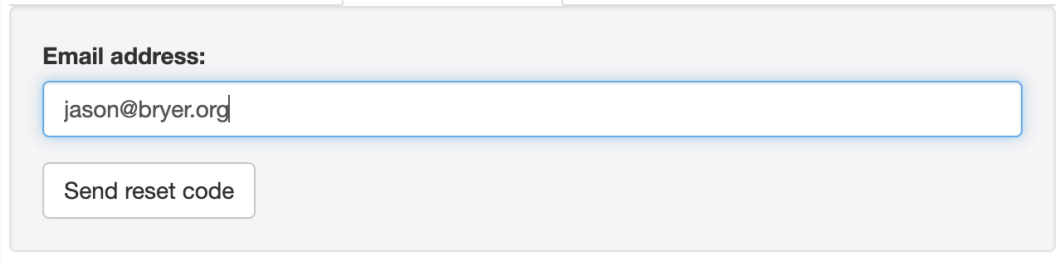
```
login::new_user_ui(id = APP_ID)
```



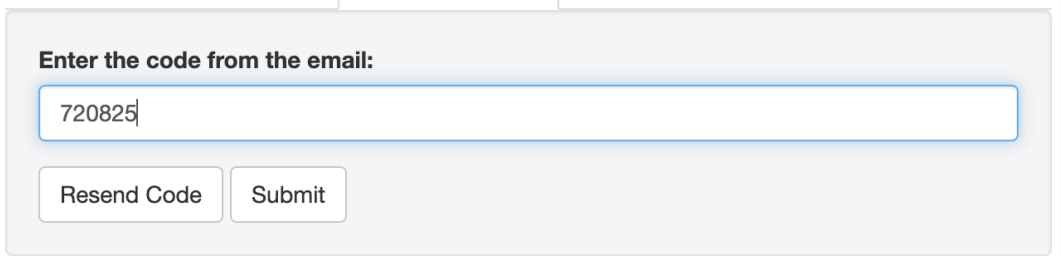
Shiny UI: Password reset

The `reset_password_ui()` function provides an interface to reset passwords. It will display differently depending on what step the user is on.

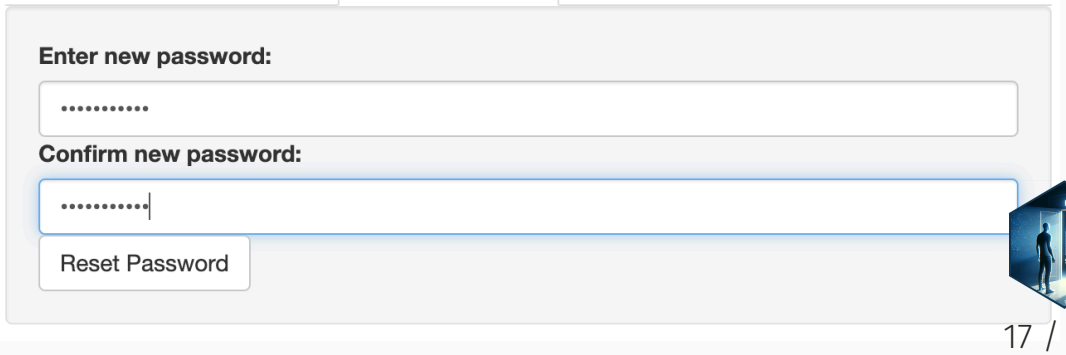
Step 1: Enter email address. If the email address is in the database then they will proceed to step 2.



Step 2: Enter the code sent to the email address (the default is a random six digit code).



Step 3: Enter a new password if the code in step 2 is entered correctly.



Adapting your UI to login state

There are two functions to assist with adapting your user interface based upon the users' authentication state: `is_logged_in()` and `is_not_logged_in()`. The `id` parameter is required, all other parameters are displayed or not depending on the user state.

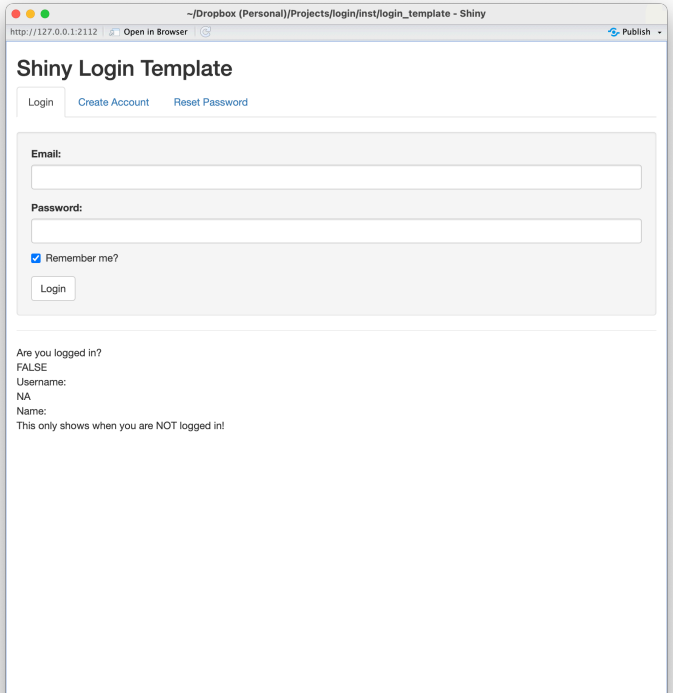
```
login::is_logged_in(
  id = APP_ID,
  div("This only shows when you are logged in!")
)
```

```
login::is_not_logged_in(
  id = APP_ID,
  div("This only shows when you are NOT logged in!")
)
```

Demos

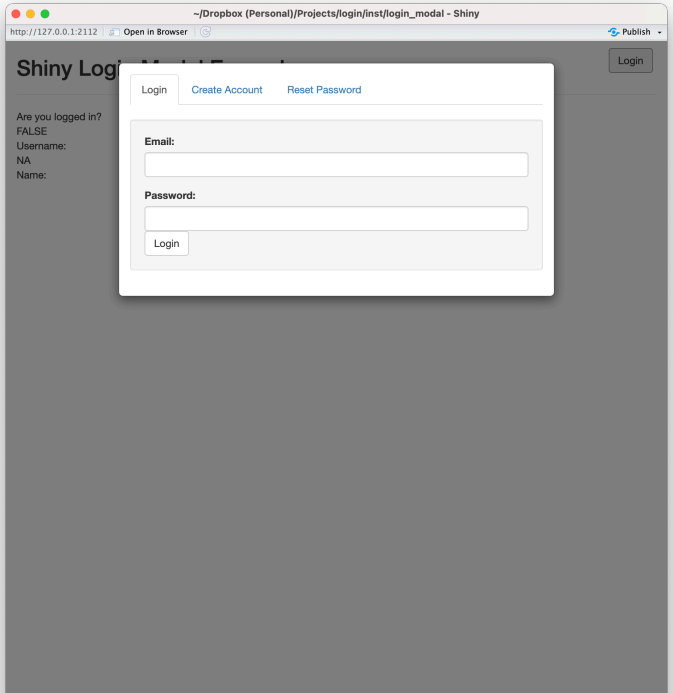
Simple template where the login UI is incorporated in your page.

```
shiny::runApp(paste0(find.package('login'),
                     '/login_template/'), port = 2112)
```



Template where users login through a modal dialog.

```
shiny::runApp(paste0(find.package('login'),
                     '/login_modal/'), port = 2112)
```



Questions?

Thank You!

jason.bryer@cuny.edu
@jbryer
@jbryer@vis.social
jbryer.github.io/login/
linkedin.com/in/jasonbryer/