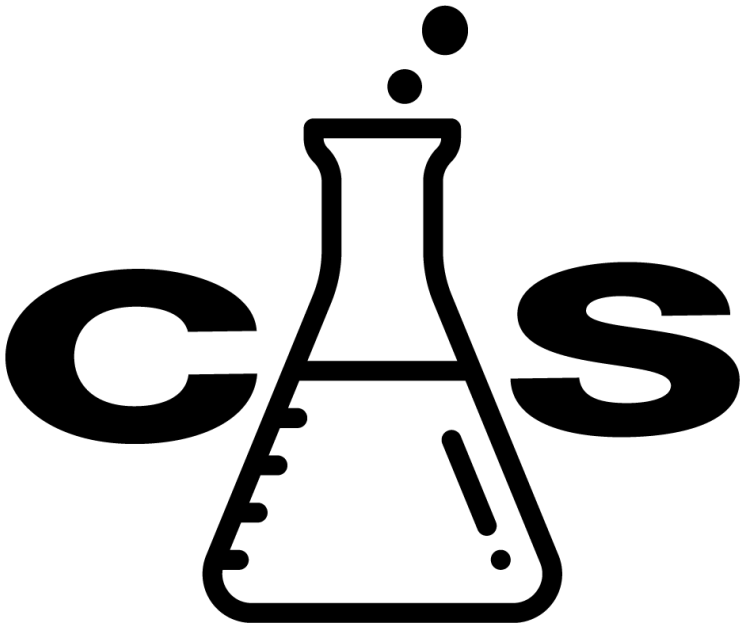


CS Labs – Web

Software Detailed Design

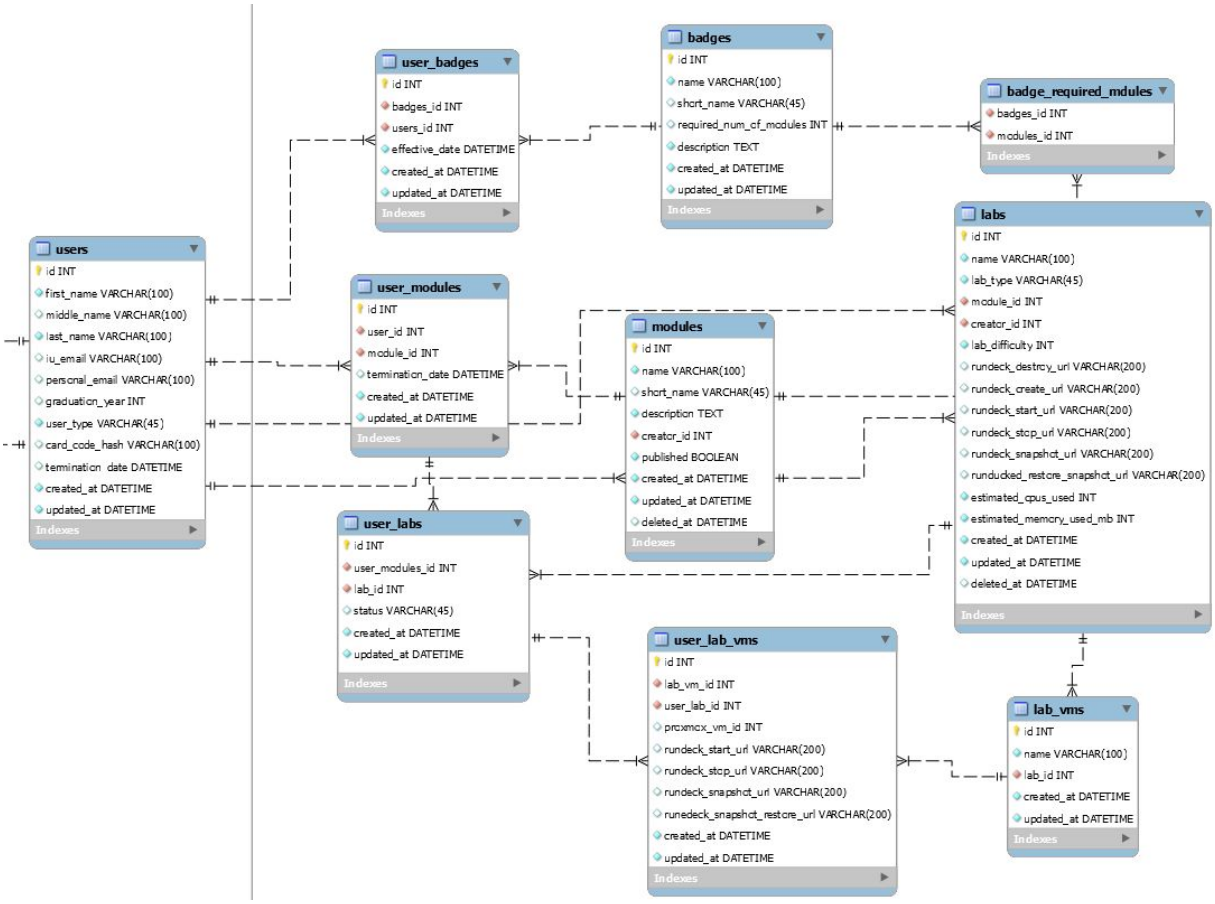
Group Leader: Jason Gallavin

Members: Barel Musenga, Chase Benston

Oct 27, 2019

Table of Contents

Data Design



Above is the ER Diagram that describes the structures in our application. Classes will be created that represent these tables in the C# HTTP Api. Typescript Interfaces in the React frontend will be created to also match these tables. Where there is a relationship, a List property exists on the C# class.

The main entity of this system is a module. A module contains a list of labs that can be completed like chapters in a book. Each lab contains a list of virtual machine definitions that should be used when the lab is started. Badges can be awarded when certain modules and other requirements are met. A Badge has an attribute called required_num_of_modules that is set if the badge is awarded when a certain amount of modules are completed. The Badge entity also has a

relation to badge_required_modules that allows a badge to specify which module(s) a user has to complete in order to be awarded.

The next larger concept in the system is the user. The user has many attributes shown in the diagram that are mostly used by the infrastructure team. Users relate to user_badges which are awarded badges that a user owns. The user also has user_modules, user_labs, user_lab_vms, which are instantiations of the module. These instantiation track progression, and lab environment state. For data structures dictionaries are used on the frontend application to quickly look up entities by Id. The backend uses Lists to store relations since all of the hard work is handled by the database.

Architecture Design

The architecture of the system consists of several parts. The front end built using React in Typescript will serve an application running in the browser that connects the backend using AJAX requests. JSON is used as the transport format. Both the frontend and backend translate to and from JSON and data objects. Using HTTP as the protocol to transmit the JSON, the frontend and backend can communicate allowing the data to display.

The C# backend includes a Rundeck API interface that also uses JSON and HTTP to communicate. The Rundeck jobs provide the following actions for the system, Start a VM, Stop a VM, Clone a new VM from a template, and destroy a VM. When the user presses the button to start the lab, that action is translated into the instantiation of a lab using multiple rundeck API calls. A small python command line interface is then called to invoke Proxmox API calls to carry out the actions. This design decision was made to create easy centralized tracking of tasks ran in the system.

Interface Design

Rundeck Api Interface

- Run job

  Runs a job via an ID. This is used to Execute commands on the Proxmox API.

- Run job and get output

Runs a job and gets the output. This is useful for commands that require a response. Commands like Create VM via template require a response of the ID of the VM so the database can track the VM.

- Connect with Host, username, and password

  The software interface requires the host, username, and password to connect to the given rundeck host.

Proxmox API Interface

- Get VM status

Retrieves the power status of the VM. This is used to display the power indicator on the front end.

- Create VM via template

When a user starts a lab, this command is executed to create a new vm from a template vm and return the ID so the VM can be deleted later on.

- Shut down VM

  This command is executed to shutdown a VM given it's ID. This will be called if the user closes the browser.

- Start up VM

  When a user starts a lab, this is called after creating the VM. The user can also manually start the VM from the frontend.

- Destroy VM by ID

When a lab should be removed. This command is issued to remove the VM. This will serve to save server space when the user is finished.

- Get Ticket

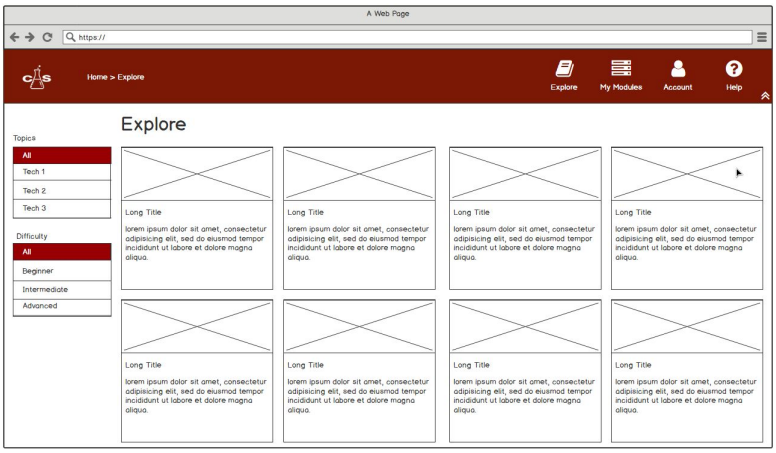
  For the frontend to be able to display a VM in a webpage, noVNC is used with a websocket. To authenticate that websocket, a ticket is required which can be retrieved.

- Connect Websocket

After the ticket is retrieved a websocket connection is requested from the frontend which is then proxied to proxmox. The websocket can then be used securely to interact with the VM.

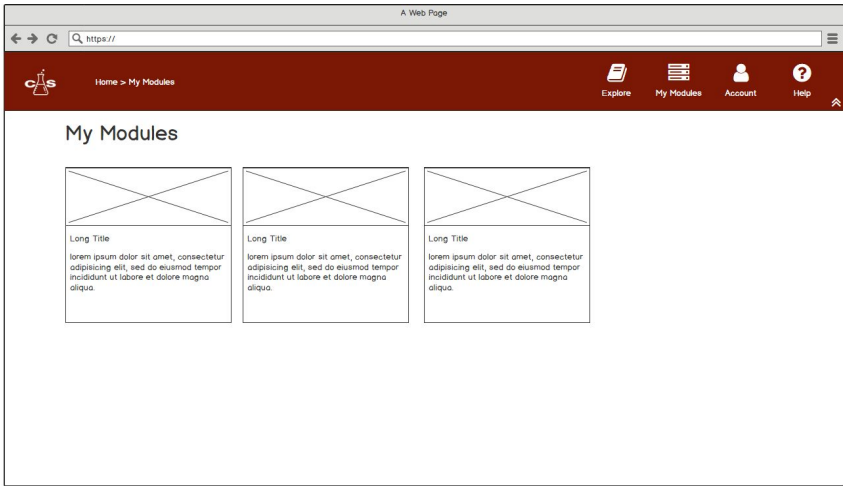
Front End Interface

The front end supports the following web browser IE 11, Edge, Firefox, Chrome, Safari. Below are the interface design of the front end.



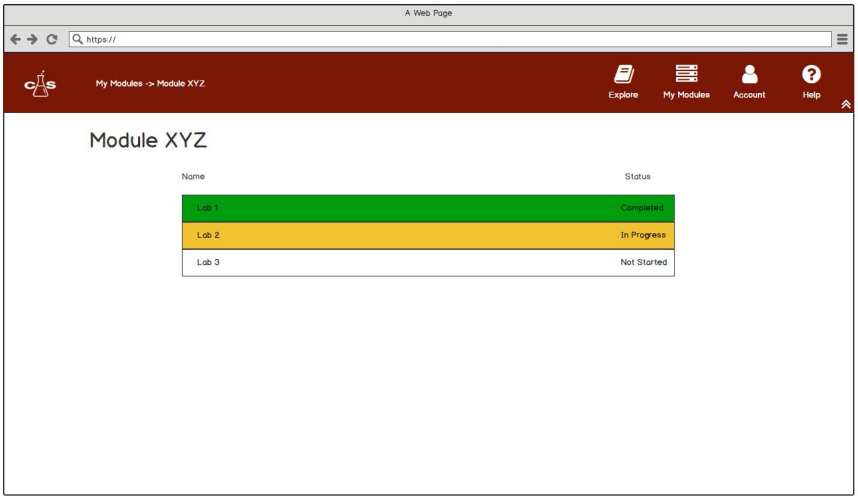
Here is where the user will select a module they are interested in.

My Modules Page



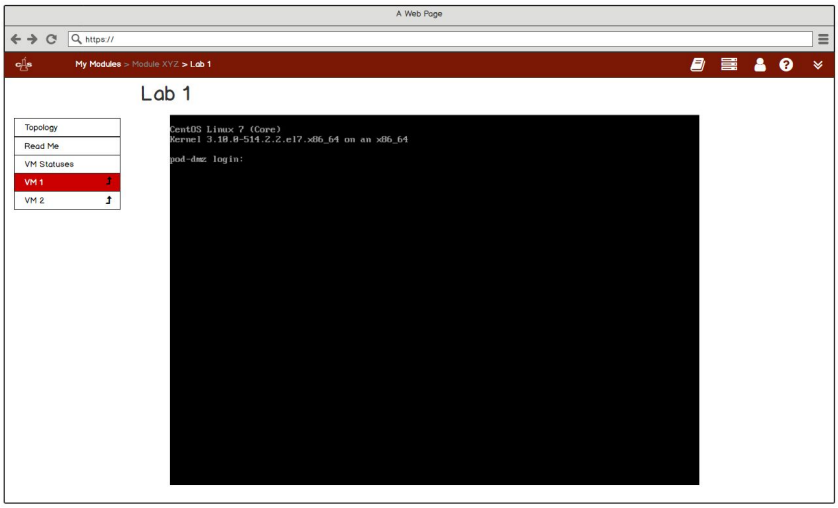
Here the user can browser modules they have started.

Module Page



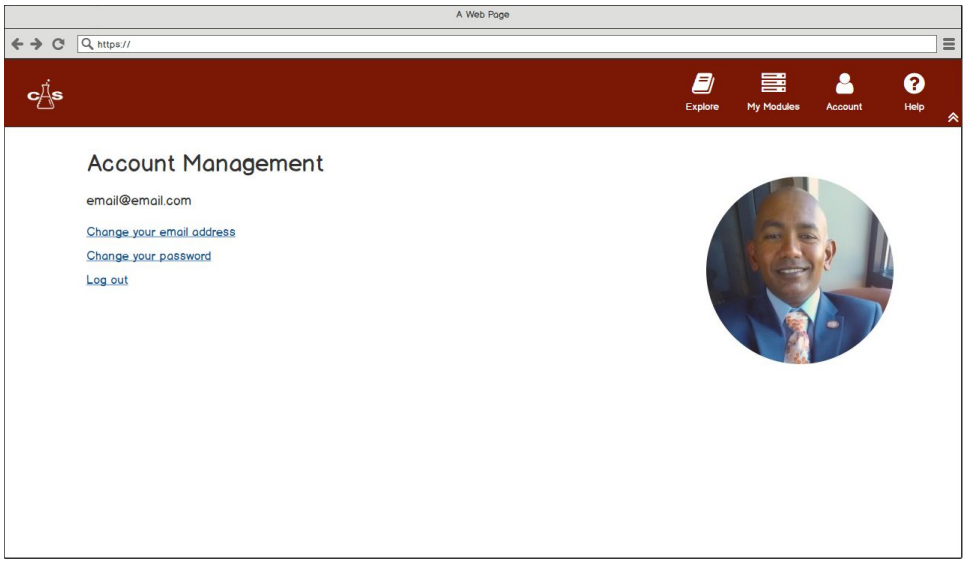
The user will select which lab they want to start. This page also shows your progression through the module.

Virtual Machine Page

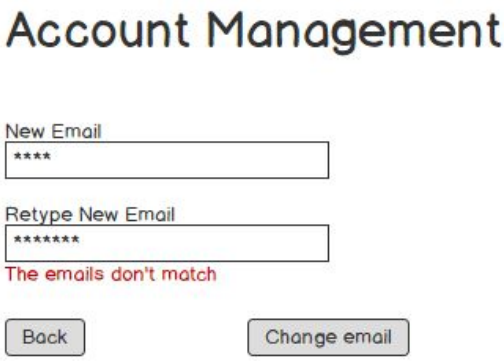


Here the user can interact with virtual machines and complete the lab.

Account Management Page



Here the user can change their email address, and password. When you press either link, the respective content shows in place. The user can upload a new photo to replace the current one.



Procedural Design

Create Lab

When the user starts a lab, the create lab procedure will start. This procedure will clone the template VMs for that lab and start the VMs.

Input

The template VM Ids as a list of arguments.

Output

The list of instance ids for the cloned vms.

Component

Converts input to output by looping through the id's and calling the clone template command on proxmox. Each call to proxmox will return the instance id.

Constraints

Must be able to start a lab in less than a minute. This design is constrained to the proxmox's api so for optimal performance, multithreading might be needed to make parallel calls.

Process

```
templateIds= getTemplateIdsFromArguments()

instanceIds = []

foreach(templateIds as templateId):

        instanceIds[] = cloneTemplate(templateId)

print(JSON.dump(instanceIds))
```

Shut Down Lab

When a user is finished with a lab, or is finished for the day. The lab should shut down to save memory and cpu resources.

Input

The input is the lab id given by the frontend when the user presses the shutdown button. This will send a list of vm ids to proxmox.

Output
There will be a string response of “success”.

Component

Converts input to output by looping through the vm ids and calling the proxmox shutdown api command for each vm id.

Constrains

A lab should be able to shutdown in around 30 seconds. Parallel programming can be used to speed this up.

Process

```
instanceIds= getInstanceIdsFromArguments()

foreach(instanceIds as instanceId):

        shutdownVm(instanceId)

 print(“sucess”)
```

Startup Lab

After a lab has shut down, the user might need to start again where they left off. This procedure turns a lab back on.
Input

The VM ids are given as arguments. These will be the VMs to start.

Output

There will be a string response of “success”.

Component

Converts input to output by looping through the vm ids and calling the proxmox start api command for each vm id.

Constraints

Must be able to start a lab in less than a minute. This design is constrained to the proxmox's api so for optimal performance, multithreading might be needed to make parallel calls.

Process

instanceIds= getInstanceIdsFromArguments()

foreach(instanceIds as instanceId):

    startVm(instanceId)

print("sucess"))

Destroy Lab

When a user is permanently finished with a lab, it should be destroted to save memory and cpu resources.

Input

The input is the lab id given by the frontend when the user presses the destroy button or if the user has left the lab unattended for a week. This will send a list of vm ids to proxmox to destroy.

Output

There will be a string response of "success".

Component

Converts input to output by looping through the vm ids and calling the proxmox delete api command for each vm id.

Constrains

A lab should be able to be destroyed in around 30 seconds. Parallel programming can be used to speed this up.

Process

instanceIds= getInstanceIdsFromArguments()

foreach(instanceIds as instanceId):

```
        destroy(instanceId)

print("sucess")
```

Register

For a user to gain access to the system, they will need to register for an account.

Input

- First Name
- Last Name
- School Email
- Personal Email
- Graduation Year
- Phone Number
- Password
- Confirm Password

Output

A base64 encoded JWT token to be used on all api requests.

Component

A user is created in the database with the hashed password and the attributes assigned. A JWT token is then generated and sent to the client.

Constrains

This process should be very quick, less than 100ms response time. Either the School email or personal email are required. Either can be used to login to the account. The remaining required fields are First Name, Last Name, Password, and Confirm Password

Process

```
if(request.Password != request.ConfirmPassword)

   Return "Passwords do not match"

if(!request.FirstName)
```

Return “First Name is required”

if(!request.LastName)

Return “Last Name is required”

if(!request.SchoolEmail && !request.PersonalEmail)

Return “School email or Personal Email is required”

user = register(request.all())

Return user.makeToken()

Login

For a user to gain access to the system once they are registered, they will need to log into their account

Input

- Email
- Password

Output

A base64 encoded JWT token to be used on all api requests. Validation message on error.

Component

The password is hashed and then searched in the database along with the email. If a user object is found, a JWT token will be generated to be served as an authentication token.

Constrains

This process should be very quick, less than 100ms response time. Either the School email or personal email are required.

Process

if(!request.Password)

Return “Password is required”

```
if(!request.Email)
    Return “Email is required”
user = DB
.Where(u => u.PersonalEmail == request.Email || u.SchoolEmail == request.Email )
.Where(u => u.Password == hashed).First()
if(!user)
    Return “Incorrect credentials”
Return user.makeToken()
```

Contribution Breakdown

	Barel	Chase	Jason	Grand Total
10/13/2019	5:30:00	1:00:00	12:25:00	18:55:00
10/20/2019			9:45:00	9:45:00

Key Personnel Information

Jason Gallavin – Project leader, contributes by creating reports, deciding architecture and larger project decisions.

Barel Musenga – Full Stack Developer, contributes by implementing the front end and back end, and parts of the lab automation.

Chase Benston – Full Stack Developer, contributes by implementing the front end and back end, and parts of the lab automation.