Sugarizer School Portal

About Me

Name: Kumar Saurabh Raj

Email: kumarsaurabhraj.sr@gmail.com

Github: **https://github.com/ksraj123**

Location: Patna, Bihar, India

Time Zone: UTC +5:30

Working Hours: 9:00 AM to 11:00 AM, 11:30 AM to 1:30 PM, 3:00 PM to 6:00 PM, 8:00PM to 11:00 PM. Working hours can be adjusted if required and I have no issues working outside my normal working hours.

Degree: Bachelor of Technology(B. Tech)

Major: Computer Science and Engineering

Institute: Bhagalpur College of Engineering, Bhagalpur

Motivation to take Part in Google Summer Of Code?

I feel like I have learnt more in the past few months preparing and contributing for GSOC than I would have learnt the entire year before that. Contributing to open source is a great way to grow as a developer and GSOC is a perfect platform which connects students willing to learn with experienced mentors. I like challenges, I sometimes spend hours figuring out what might be the reason for a bug or how to make something work and at the end of the day, I love that feeling when I figure it out even if I don't, I get to learn so much through it. I love it and I want to keep doing this.

Why did you choose Sugarlabs?

As a developer I want my code to matter. I want to code to be a part of something which empowers people and makes an impact. I feel that a child has unlimited potential and

he can grow up to be anything. Sugarlabs softwares are used by children and educators. It feels nice to think that what I am developing will be used by a child somewhere.

Why do you want to work on the particular project?

I have some prior experience working the technologies used in this project and reading through all of the ideas I felt like this one provides the greatest opportunity to learn. Also I love sugarizer, I think it would have been fun to have it as a child and this project is basically about bringing sugarizer to more people, that gets me excited.

What are your expectations from us during and after successful completion of the program?

I expect to be mentored by the assigned mentors and the rest of the community. I expect the mentors to review my work and provide feedback or suggestions. Mentors have a lot more experience than me so I would like to them to guide me if I get stuck somewhere.

Skills: Javascript, React, Node.js, express, MongoDB, Redis, Docker, Ansible, Kubernetes, Helm, Google Cloud Platform, Google Kubernetes Engine

Work Done on the Idea So Far

- A helm chart created for sugarizer-server which installs sugarizer-server onto the Google Kubernetes Engine. An nfs server is created which uses gce persistent disks to provide data persistence or statefulness. Links to a working deployment on GKE using this chart can be found in the Readme section of the repository. Link :- https://github.com/ksraj123/sugarizer-server-chart

- An Ansible package is created which creates a new cluster on Google Kubernetes Engine and installs our application itself and its dependencies. Link :- https://github.com/ksraj123/sugarizer-server-k8s-ansible

- Backend for a functional super admin console is created using which the super admin will be able to add a new deployment, approve deployment requests,

destroy deployments, add or remove resources from a deployment or the cluster. UI and bug fixes will be completed in Early April. Once the ui is complete the demo project will be deployed on GKE and links to the same will be provided in the Readme section of the repository.
Link :- https://github.com/ksraj123/sugarizer-school-portal

Contributions to Sugarlabs

Pull Requests

- FullScreen UnFullScreen Button added to Video Viewer activity
- Improvements to fullscreen functionality
- [Issue #680 fix] Added replay functionality to Abecedarium activity
- Issue #675 fixed Improvements in responsiveness of Pomodoro activity
- tutorial with localization added to stopwatch activity
- Fixing issues in tutorial of game of life activity
- [Issue #686] Simon Mode added to TamTamMicro activity
- FullScreen Unfullscreen button added with resizing functionality
- Added basic circular scrolling to restricted mode sugar spiral
- fixed redundant scrollbar and added space below submit in sub activities
- Tutorial added to Tank Operation activity

Issues

- Abecedarium Activity not working on mobile resolutions in file:///
- Improvements to navigation in Restricted mode of Spiral
- Proposal for new sugarizer activity - I know my world

- Position of Toolbar buttons wrong at mobile resolution
- Issues in game of life activity
- No space below submit button in multiple sub activities
- Responsiveness issue in Pomodoro Activity

Other projects:-

- **myUniversityHackathon:-** It is a web application developed using Node.js, MongoDB and jQuery. It has been used as a template by a few universities in my region to build their own hackathon websites. We conducted a hackathon in my university with this website and deployed it on the Google Cloud Platform.
link - https://github.com/ksraj123/myUniversityHackathon

- **Multiplayer Chess Sugarizer Activity:-** A multiplayer chess activity was developed for sugarizer using Vue and presence to provide multiplayer support. Completion for this activity was also a qualification task for some Ideas but not for the particular Idea I am applying to, hence I did not submit a PR. I also completed the **PAWN activity** to learn about how sugarizer works under the hood.
chess activity - https://github.com/ksraj123/sugarizer/tree/someWorkingChess
pawn activity - https://github.com/ksraj123/sugarizer/tree/pawnActivity

- **RAW12-to-bitmap-AVI**:- C++ - Sensors used in digital cameras lack the ability to capture color images as they lack the ability to distinguish how much of each color they are receiving. To capture color images a filter is placed above the sensor which permits only a particular color light at each sensel but as a result of this, each sensel has intensity values of only one channel which is not sufficient to produce a color image. Hence different demosaicing algorithms are used to obtain full-color RGB images from RAW images.
link - https://github.com/ksraj123/RAW12-to-bitmap-AVI

About the Idea -

Sugarizer School Portal is a new tool to help schools interested in hosting and managing their own sugarizer deployment themselves without much technical knowledge. The sugarizer-server will be deployed as a web application onto a kubernetes cluster which can be accessed over the internet as a SaaS (Software as a service). The super administrator will create the cluster and approve deployment requests. The Sugarizer School Portal comprises mainly two parts the Super Admin console and the School Console. The schools can request for a deployment for themselves with the school console. The super admin (owner of the cluster) will interact with the Super Admin Console to approve deployment requests and monitor resources or change resource allocations. The idea behind this project is to make sugarizer accessible to more institutions. So any task which might require much technical know how like interacting with the cluster, creating new nodes, scaling etc is abstracted through the application so that it can be done by a few clink on the user interface. The application is designed to be user friendly and economic to host so that any interested organization like a charity, ngo, schools, orphanage etc can have their own cluster and enable other institutions around them to use sugarizer.

Getting Sugarizer-Server Kubernetes Ready

I have created a helm chart for installing sugaizer-server. The current approach is very non-invasive and as little of the existing code is changed as possible. However many changes and adaptations are needed to make the sugarizer-server work properly when deployed onto a kubernetes cluster. Some of the issues observed with sugarizer-server are -

- **Handling Sessions across multiple pods**

  When there are more than one instances of the sugarizer-server in the cluster then on logging in the user gets redirected to the login page. Currently sugarizer-server does not use any session store and in express-session the default value of store is MemoryStore i.e. the sessions are stored in memory. When there are more than one pods then the request may be routed to any pod and one pod is not aware of a session stored in another pod’s memory. This probably might be the reason for this issue and could be fixed by using a session store common to all pods on a central database.

- **Handling upgrades**

  Currently in sugarizer-server not much data is actually present inside the container, various links mount directories in the container to volumes on the host. This would not work well in a kubernetes environment as pods and nodes are ephemeral, they keep on getting destroyed and regenerating. Also if a new version of sugarizer-server is released then there won't be a simple way to upgrade the entire cluster but to upgrade every storage location which has the application's code individually. Hence an image of sugarizer-server will be created which will contain all the Node.js application code within it so that in case of a future release all the pods can be easily updated in rolling fashion using helm upgrade command by upgrading the image to the new image.

- **Handling Statefulness and Database**

  Data persistence is an issue in any kubernetes deployment. Currently sugarizer-server stores its database files onto the host which would again not work well on kubernetes as the request can be routed to any pod and database files inside the pods will not be in sync. Various approaches to handle this have been discussed in the following sections.

Sugarizer Server Chart - Implementation

In this section we will go over some implementation details of the sugarizer server chart and how sugarizer is made to work in kubernetes.

```
#sugarizer-chart/namespace.yaml

kind: Namespace
apiVersion: v1
metadata:
 name: {{.Values.school.Id}}
 labels:
   name: {{.Values.school.Id}}
```

All the Kubernetes objects of our chart will be created inside the namespace which will be passed in from the command line using the --set flag. The command will actually get executed by an ansible playbook executed by our application. So ultimately a unique school Id will be generated for each school which will be its namespace.

```
# templates/server_deplyment.yaml - deploys the sugarizer server
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: school-deployment
 namespace: {{.Values.school.Id}}
spec:
 replicas: {{ .Values.server.replicas }}
 selector:
   matchLabels:
       app: {{ template "sugarizer-chart.fullname" . }}-pod
 template:
   metadata:
       labels:
         app: {{ template "sugarizer-chart.fullname" . }}-pod
         school: {{.Values.school.Id}}
   spec:
     volumes:
       - name: my-pvc-nfs
         persistentVolumeClaim:
           claimName: nfs
     containers:
     - name: sugarizer-server
       image: {{ .Values.server.image }}
       volumeMounts:
       - name: my-pvc-nfs
         mountPath: /sugarizer-server
         subPath: sugarizer-server
       - name: my-pvc-nfs
         mountPath: /sugarizer-client
         subPath: sugarizer-client
       command: ["/bin/sh", "-c"]
       args:
       - apt-get update; apt-get install sudo; sudo apt install git -y;
         {{ .Values.client.setupCommands }}
         {{ .Values.server.setupCommands }}
         sh ./add-admin.sh admin password http://127.0.0.1:80/auth/signup;
       env:
       - name: NODE_ENV
         value: docker
       - name: MONGO_URL
```

```
          value: mongodb://{{ template "mongodb.fullname" .
        }}-service.{{.Values.school.Id}}.svc.cluster.local:{{ .Values.database.port }}/
        ports:
        - containerPort: {{ .Values.server.httpPortPod }}
          name: http
        - containerPort: {{ .Values.server.presencePort }}
          name: presence
```

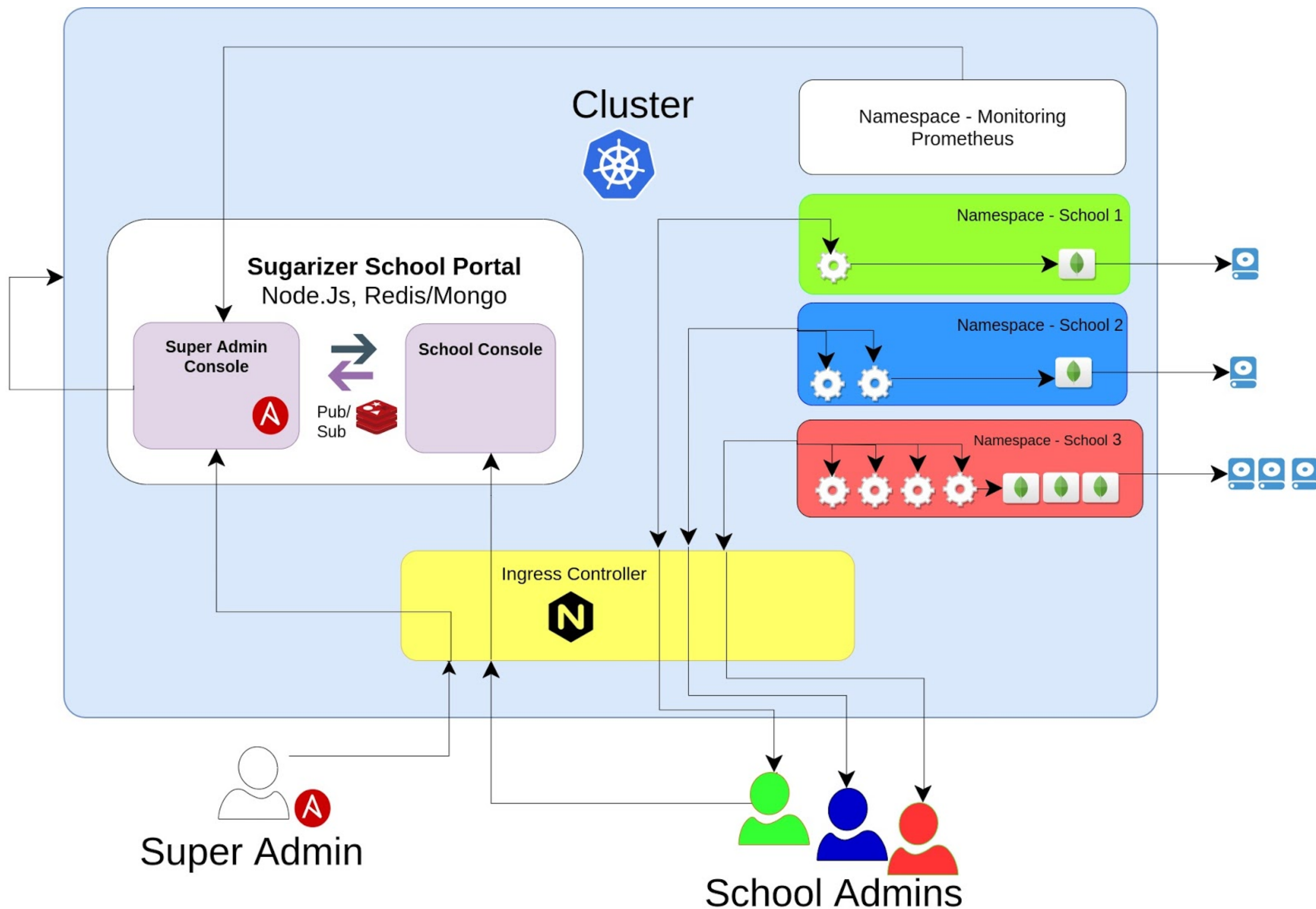
The most important thing in our server-deployment file is the environment variable MONGO_URL we do not have the mongo container in the same pod as the sugarizer-server container so to connect to a mongodb pod on our cluster the using a clusterIp service the url should be in the form of <service_name>.<namespace>.svc .cluster.local. The code in sugarizer-server has to be modified to use the environment variable in instead of localhost url if it is defined.

Here is what the manifest file for deploying mongoDb and the clusterIP service for accessing it from our sugarizer-server pods might look like -

```
# templates/mongo-deployment.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
 name: {{ template "mongodb.fullname" . }}-deployment
 namespace: {{.Values.school.Id}}
spec:
 replicas: 1
 selector:
   matchLabels:
     app: {{ template "mongodb.fullname" . }}-pod
 template:
   metadata:
     labels:
       app: {{ template "mongodb.fullname" . }}-pod
   spec:
     volumes:
       - name: my-pvc-nfs
         persistentVolumeClaim:
           claimName: nfs
     containers:
     - name: mongodb
```

```
       image: {{ .Values.database.image }}
       volumeMounts:
       - name: my-pvc-nfs
         mountPath: /data/db
         subPath: sugarizer-server/docker/db
       command: ["/bin/sh", "-c"]
       args: ["mongod --repair; mongod"]
       env:
       - name: AUTH
         value: "no"
       ports:
       - name: mongodb
         containerPort: {{ .Values.database.port }}
         hostPort: {{ .Values.database.port }}
         protocol: TCP

---

apiVersion: v1
kind: Service
metadata:
 name: {{ template "mongodb.fullname" . }}-service
 namespace: {{.Values.school.Id}}
spec:
 ports:
 - port: {{ .Values.database.port }}
   targetPort: {{ .Values.database.port }}
   protocol: TCP
 selector:
   app: {{ template "mongodb.fullname" . }}-pod
```

Design



- Different components of our application will live in different namespaces in the same cluster. Resource limits will be placed on namespaces to determine what part of entire resources available to the cluster can a component use.

- The entire application has three main components:-
    - Sugarizer School Portal
    - Sugarizer Deployment Namespaces
    - Monitoring namespace

- The Super admin has the following jobs:-
    - Setting up the cluster for the first time
    - Approving Deployment Requests or Create new Deployment
    - Deleting or Destroying Deployments
    - Changing resources allocated to a deployment

    - Adding or Removing resources from a cluster
    - Adding new Clusters
    - Monitor sugarizer deployments and Cluster
    - Add more super administrators.

- The school admin has the following jobs:-
    - To request for a new deployment on the School console.
    - To create accounts for more school admins.

- The sugarizer school portal too will live in its own namespace on the cluster. it will have two parts:-
    - Super admin console :- The super admin console abstracts cluster management, resource allocation and monitoring by providing options in the user interface to perform those tasks.
    - School Console :- Using the school console the school admins will be able to request new deployment for their school. They could also login to view information about their deployment, request more resources, add more school admins for their deployment and request contact super admins for support.

- Let's briefly go over how the application will work :-

    - The Super Admin will do the first time setup by executing an ansible package which will automatically do the following tasks: -
        - Create a new kubernetes cluster
        - install the sugarizer school portal application on it
        - Install NGINX Ingress Controller
        - install prometheus in a new namespace

      It is discussed in greater depth in the following sections

    - The super admin console will execute various playbooks using the node-ansible package to connect to the cluster and perform operations on it depending upon super admin's interaction with the UI. Technical details of how this will be implemented is discussed in the following sections.

    - Requests from one school will be routed to service of that school's namespace by the ingress controller.

    - Each school's namespace will contain pods running sugarizer-server and the database. The number of pods will scale automatically to make best use of available resources for that namespace. This will ensure high availability.

- Pods are ephemeral and the data inside is destroyed as well when the pod is destroyed. To make the data persistent, In the demo application an nfs server has been created using gce persistent disks. Traditional nfs servers provide ReadWriteMany access but they do not make sense economically for small deployments. Gce persistent disks are cheap and better suited for small deployments but they support only ReadWriteOnce or ReadOnlyMany access mode. So an nfs server is deployed in the cluster which stores data into persistent disks to get ReadWriteMany Access mode. Various approaches to handle statefulness have been discussed in the following sections. Out of the discussed approaches using stateful stateful sets or gce persistent disk nfs server seem to be good choices.

- The super admin console will fetch metrics from the prometheus HTTP API and display graphs to helm monitor various namespaces and the cluster itself. The super admin will decide to adjust resource limits of a deployment or add resources to the cluster.

- The super admin can add more super admins and there can be many school admins. So the need for a redis based real time publish subscribe relationship will be explored.

First Time Setup

First Time Setup refers to the steps the Super Administrator has to take to set up the kubernetes cluster and deploy the Sugarizer School Portal application on it. The super admin will be able to perform tasks like creating new sugarizer-server deployments, approving deployment requests, managing resources for existing deployments etc on the Sugarizer Server Portal application.

The schools can request for deployments which will be done on the cluster provided by sugarlabs but they should be able to make it work on their own cluster if they want to. Apart from this it's in the spirit of open source to enable anyone like a university, charity or a similar organization to make their own cluster and help other schools to host their sugarizer-server deployment with as little technical knowledge as possible. Hence the First Time Setup and the Super Admin console will be made such that it can be handled with very basic technical knowledge.

An ansible package has been created which will install google cloud sdk, create a new kubernetes cluster, connect to the cluster, install helm 3 and install the helm chart for

Sugarizer School Portal and will provide a link using which the the web ui of the super admin console could be accessed. A shell script will be created for installing all the dependencies for the ansible package.

The super administrator will have perform the following tasks before executing the playbook -

- Start a project on Google Cloud Platform and note down details like project id, zone etc.
- Enable kubernetes API for the project.
- Generate a service account file (account.json) and paste it into the root directory of the playbook.
- Execute a script which will install dependencies like python, pip, google-auth, ansible etc.
- Set Values in the values file of the ansible package.

Here is how the directory structure of the first time setup ansible package looks like -

```
Sugarizer-School-Portal-Setup
├- host_vars
│  └- localhost
├- roles
│  ├- create-cluster
│  ├- role-install-gcloud
│  ├- connect-to-cluster
│  └- install-helm
├- school-portal-chart
├- account.json
├- create-new-cluster.yml
├- install-gcp-sdk.yml
├- install-helm.yml
├- install-school-portal-chart.yml
└- site.yml
```

Lets go over the important aspects of the package-

```
# host_vars/localhost

auth_kind: serviceaccount
service_account_file:  # serviceAccountKey file for verifying and identifying google account
project:                             # Enter the Google Cloud Platform project id here
cluster_name:                        # enter name of cluster, suggested value - name of
school
initial_node_count:                  # number of nodes in the cluster initially
```

```
username:                            # username for accessing cluster master endpoint
password:                            # password for accessing cluster master endpoint
zone:                                # set the zone in which the cluster is to be created
machine_type: n1-standard-1
disk_size_gb: 100
# gke node pool
npname: gkeclus-pool
np_initial_node_count:
```

Here is what the master playbook looks like -

```
# Master playbook - should be executed for first time setup

- import_playbook: create-new-cluster.yml

- name: Installing the Google Cloud Sdk
 import_playbook: install-gcp-sdk.yml

- name: Installing Helm
 import_playbook: install-helm.yml

- name: Installing Sugarizer School Portal Helm Chart
 import_playbook: install-sugarizer-chart.yml
```

install-gcp-sdk playbook makes use of a standard role provided by ansible - https://github.com/ansible/role-install-gcloud and the create-new-cluster makes use of standard modules provided by google gcp_container_cluster and gcp_node_pool

```
# roles/create-cluster/tasks/main.yml

- name: create a cluster
 gcp_container_cluster:
     name: "{{ cluster_name }}"
     initial_node_count: "{{ initial_node_count }}"
     master_auth:
       username: "{{ username }}"
       password: "{{ password }}"
     node_config:
       machine_type: "{{ machine_type }}"
       disk_size_gb: "{{ disk_size_gb }}"
     zone: "{{ zone }}"
     project: "{{ project }}"
```

```
    auth_kind: "{{ auth_kind }}"
    service_account_file: "{{ service_account_file }}"
    scopes:
      - https://www.googleapis.com/auth/cloud-platform
    state: present
 register: cluster

- name: create a node pool
 gcp_container_node_pool:
    name: "{{ npname }}"
    initial_node_count: "{{ np_initial_node_count }}"
    cluster: "{{ cluster }}"
    zone: "{{ zone }}"
    project: "{{ project }}"
    auth_kind: "{{ auth_kind }}"
    service_account_file: "{{ service_account_file }}"
    scopes:
      - https://www.googleapis.com/auth/cloud-platform
    state: present
```

Super Administrator - Jobs - Implementation.

In this section we will look at the various jobs the super administrator will be able to perform using the super admin console and how they will be implemented technically. The admin console Node.js application interacts with the cluster by executing Ansible playbooks using npm node-ansible package. The details about the project itself project_Id, zone etc will be provided by the super administrator during the first time setup in the Ansible values file and they will be passed to the admin console Node.js application as environment variables and will eventually be stored in the database. The super administrator has the following jobs -

Approving Deployment Requests or Creating new Deployments

The super administrator will receive deployment requests from schools. He will review them and can either approve or reject them. He can also create a new deployment without any prior request. Technical implementation of both will be similar, the only difference being that the source of information which is a form in the admin console app while creating a new deployment or request by school admin in the case of approving deployments. Here is a how this might be implemented -

```
// superAdminConsole/createNewDeployments.js

var Ansible = require('node-ansible');

module.exports = function createNewDeployment (cluster, school, project){
   var config = {
       project: process.env.PROJECT_ID || project.project_Id,
       zone: process.env.PROJECT_ZONE || project.zone,
       cluster_name: cluster.name,
       schoolId: school.Id
   }
   var command = new
             Ansible.Playbook().playbook('./ansible/new-school-deployment').variables(config);
   var promise = command.exec();
   promise.then(function(result){
       // playbook successfully finished execution
       var getJson = new
                Ansible.Playbook().playbook('./ansible/get-external-ip').variables(config);
       getJson.exec().then(function(jsonRes){
           // external IP is allocated after a few minutes of successful installation
           // so few minutes pause added to external ip playbook
           var getJson = new
                Ansible.Playbook().playbook('./ansible/get-external-ip').variables(config);
           getJson.exec().then(function(jsonRes){
               var res = jsonRes.output.slice(jsonRes.output.indexOf('{'),
                jsonRes.output.lastIndexOf('}')+1);
               res = JSON.parse(JSON.parse(res).json.stdout_lines.reduce((acc, curr) =>
                acc+curr, ""));
               console.log(`External Ip = ${res.status.loadBalancer.ingress[0].ip}`);
           }).catch(function(err){
               console.log(err);
           })
       }).catch(function(err){
           console.log(err);
       })
   }).catch(err => {
       console.log(err);
   })
}
```

The above code uses the node-ansible package to execute the new-school-deployment playbook under the ansible directory in our application. When the execution is finished the promise is resolved and upon successful execution another playbook get-external-ip is executed and the external ip is retrieved from its result after some operations which will be used for accessing the newly created sugarizer-server deployment. Since it takes a few minutes for the external ip to be allocated I have added a few minutes pause to the get-external-ip playbook.

Let's have a look at new-school-deployment and get-external-ip playbooks:-

```
# Ansible/new-school-deployment.yaml
- hosts: localhost
 tasks:
  - name: connecting to cluster
    shell: gcloud container clusters get-credentials {{cluster_name}} --zone {{zone}}
--project {{project}}
  - name: creating new gcp persistent disk
    shell: gcloud compute disks create --size=10GB --zone={{zone}} {{schoolId}}
  - name: installing sgarizer-server chart
    shell: helm install {{schoolId}} ./sugarizer-chart --set school.Id={{schoolId}}
```

Values passed through the variables function of our node-ansible playbook object is equivalent to providing values from the command line using -e flag so it overrides any values from ansible values files. Script for installing google cloud sdk will be provided in the pod specification of super admin console deployment manifest so that every time a pod is created, it has google cloud sdk installed so the gcloud command can be used directly. In the demo application to handle statefulness a nfs server is created which stores data on a gce persistent disk hence it is created and the nfs-server deployment, persistent volumes, persistent volume claims and related services are created with chart installation. The school.Id will be unique for all schools and is used in the naming of many resources and all the kubernetes resources are created in that namespace.

```
# Ansible/get-external-ip.yaml
- hosts: localhost
 tasks:
  - pause:
    minutes: 5
  - name: connecting to cluster
    shell: gcloud container clusters get-credentials {{cluster_name}} --zone {{zone}}
         --project {{project}}
  - name: scaling the deployment
    shell: kubectl get service {{schoolId}}-service --namespace={{schoolId}} -o json
```

```
    register: json
  - set_fact:
     marker: $marker$
  - debug:
     var: json
```

By executing these two playbooks with different school ids new deployments can be created.

Delete or Destroy Deployments

The super admin should have the ability to delete or destroy the sugarizer deployment of a school. It can be done by executing a playbook and passing in the school id of the deployment from our super admin console application.

```
# Ansible/destroy-school-deployment.yaml
- hosts: localhost
 tasks:
  - name: connecting to cluster
    shell: gcloud container clusters get-credentials {{cluster_name}} --zone {{zone}}
--project {{project}}
  - name: deleting helm release
    shell: helm delete {{schoolId}}
  - name: Force Deleteing pods
    shell: kubectl delete --all pods --namespace={{schoolId}} --grace-period=0 --force
  - pause:
     minutes: 1
  - name: deleting gcp pd
    shell: gcloud compute disks delete {{schoolId}} --zone={{zone}} -q
```

Sometimes the pods get stuck in a terminating state so a force deletion step is included in the playbook. In our sample application the gce persistent disk is deleted after a small pause to ensure all the resources using it are completely deleted by the time its deletion is attempted to avoid any errors.

Adding new Clusters

The super administrator will be able to add a new cluster. However, the application is primarily designed to make use of a single cluster for multiple sugarizer deployments which will be separated by different namespaces. This makes managing deployments and automating tasks on them easier and is also beneficial economically. There is an incoming pricing policy change by Google according to which starting June 6, 2020 all clusters irrespective of their size will be charged a flat maintenance fee of $0.1 per

cluster per hour. However one zonal cluster will be provided free of charge. More details about this can be found here - https://cloud.google.com/kubernetes-engine/pricing
So if the application is designed to work well using a single cluster then this would be especially beneficial for schools willing to have their own cluster. To add a new cluster the following playbook will be executed by the super admin app -

```
# Ansible/scale-deployment.yaml
- hosts: localhost
 tasks:
  - name: connecting to cluster
    shell: gcloud container clusters create {{cluster_name}} --zone {{zone}}
```

Change resources allocated to a deployment

The super admin should be able to change the number of replicas of a deployment. The more replicas a school has, the more resources it will consume. Here is a sample playbook which can be executed from the super admin console application to change the number of replicas by passing in the new number of replicas and school Id.

```
# Ansible/scale-deployment.yaml
- hosts: localhost
 tasks:
  - name: connecting to cluster
    shell: gcloud container clusters get-credentials {{cluster_name}} --zone {{zone}}
         --project {{project}}
  - name: scaling the deployment
    shell: helm upgrade {{schoolId}} ./sugarizer-chart --set
         server.replicas={{scale}},school.Id={{schoolId}}
```

However this will not be controlled manually and horizontal pod autoscaling will be set up. The sugarizer deployments of different schools will live in different namespaces on our cluster and the resources will be split between them. With horizontal pod autoscaling the pods will automatically scale up or down to make desired usage resources allocated to a namespace. Resource quotas can be set up to limit the resources used by namespace. Total amount of resources in the cluster can be figured out with the type of machines (for example an n1-standard-1 machine has 1 vCPU and 3.75GB of Memory) and the number of nodes. Quotas allocated to each sugarizer deployment (namespace) will be kept track of by the app and will be adjusted as per requirement. The pods in a namespace will undergo horizontal auto scaling to utilize around 70 to 80 percent of quota limit. Auto scaling will also be performed on resources other than compute resources whose metrics will be retrieved from Prometheus.

Add or Remove resources from a cluster

On the Google Kubernetes Engine(GKE) platform the cost of the cluster is linked to the number of GCP VMs or nodes it has. So the super admin should have control over the number of nodes. GKE provides support for auto scaling nodes which can be implemented after discussion with mentors. The super admin will monitor the cluster resources and if the need be then he may add nodes to relieve the load on the cluster. However existing pods will not be automatically rescheduled on new nodes and the new nodes will be used for scheduling new pods. To balance load across the cluster and ensure better availability this should be avoided, a simple approach to handle this could be to downscale the deployments and then upscale them after the cluster is resized.

Here is a playbook which might be executed from the super admin app by passing in details about the project and cluster into the variables function of the node-ansible playbook object.

```
# Ansible/resize-cluster.yaml
- hosts: localhost
 tasks:
  - name: connecting to cluster
    shell: gcloud container clusters get-credentials {{cluster_name}} --zone {{zone}}
         --project {{project}}
  - name: resizing the cluster
    shell: gcloud container clusters resize {{cluster_name}} --node-pool {{npname}}
         --num-nodes {{new_cluster_size}}
```

Monitoring Cluster And Deployments

The super admin will be able to view utilization of resources by a specific sugarizer deployment (namespace) and utilization of resources of the entire cluster depending upon which he either adjusts resource allocation to a deployment or modify resources of the cluster itself.
Strings written in a query language called PromQL are used to extract data from Prometheus. To make this user friendly, we will use the Prometheus HTTP API and With the data retrieved either graphs can be plotted using some plotting library or Grafana can be embedded into the application which will provide beautiful automatically refreshing graphs and it integrates well with prometheus. Both the options will be explored and tested. Sugarizer Deployment of different schools will be inside different namespaces in our cluster. Resources consumed by one namespace will be managed by resource quotas. The Super Admin console app will keep track of resource quota values assigned to different namespaces (sugarizer deployments) and update them as required. Prometheus and Graphana will be used for monitoring.

Handling Statefulness - Implementation and Discussions

Approaches for statefulness with mongoDB as database :-

➔ A simple approach could be to use Google Filestore nfs and mounting the location inside the pods to it. However Google Filestore is meant to handle large file systems as the smallest possible capacity is 1 TB which is more than what most schools would need. This approach is not favourable economically as well, the price of 1GB in standard tier is $0.2/month which comes to a minimum of $200/month for storage. We could store database files from multiple schools in the same filestore volume to make this economically viable but it would still not make sense for schools interested in having their own cluster. Also the performance might suffer with multiple deployments fetching and writing data to the same volume.

➔ Another approach could be to use gcp persistent disks with StatefulSets. In our StatefulSet multiple pods will be linked together in a single master multi slave configuration. The slaves will replicate data from the master. Read operations can be done from either master or slave but write operations are done only to the master. If the master fails then one of the slaves takes its place through the process of election. Ssd based gce persistent disks will be linked to each pod for storing its data. The price per GB per month of a ssd gce persistent disk is $0.17 so if we have 9 pods in our StatefulSet each linked to 10GB ssd persistent disk the monthly cost would be only around $15.

This StatefulSet will go along with a StorageClass which will create ssd gce persistent disk voluments upon request from the StatefulSet. A headless service will also be created so connection can be made individually to the pods.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
 name: ssd-gcePd
provisioner: kubernetes.io/gce-pd
parameters:
 type: pd-ssd

---
```

```yaml
apiVersion: v1
kind: Service
metadata:
 name: mongodb
spec:
 ports:
 - port: {{ .Values.database.port }}
   targetPort: {{ .Values.database.port }}
 clusterIP: None
 selector:
   app: mongodb
```

There is a standard helm chart available from helm itself for implementing this approach which can be tweaked to suit our needs.
https://github.com/helm/charts/tree/master/stable/mongodb-replicaset

Here is what a simple Mongo StatefulSet for this approach might look like -

```yaml
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
 name: mongodb
spec:
 serviceName: "mongodb"
 replicas: {{ .Values.database.replicas }}
 template:
   metadata:
     labels:
       app: mongodb
   spec:
     terminalGracePeriodSeconds: {{ .Values.database.terminalGracePeriodSeconds}}
     containers:
       - name: mongodb-pod
         image: {{ .Values.database.image }}
         command:
           - mongod
         args:
           - replSet={{ .Values.database.replicaSetName }}
         ports:
           - containerPort: {{ .Values.database.port }}
         volumeMounts:
           - name: mongodb-persistent-storage
             mountPath: /data/db
```

```
VolumeClaimTemplates:
- metadata:
    name: mongodb-persistent-storage
    annotations:
      volume.beta.kubernetes.io/storage-class: "ssd-gcePd"
    spec:
      accessMode: ["ReadWriteOnce"]
      resources:
        requests:
          storage: {{ .Values.database.gcepdSize }}
```

➔ Another approach could be to create a seperate NFS file server for each school on the cluster using gce persistent disks. This approach has the benefits of NFS file servers like ease of use while still keeping the costs very low as it uses gce persistent disks. I have used this approach in the demo application.
Here is how the deployment for nfs server might look like :-

```
# before this chart is installed a a gce persistent disk will be created by
# the Ansible playbook with the following command
# gcloud compute disks create --size=10GB --zone=us-east1-b <Name of gcePd>
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
 namespace: {{.Values.school.Id}}
spec:
 replicas: {{ .Values.nfs-replicas}}
 selector:
   matchLabels:
     app: nfs-server
 template:
   metadata:
     labels:
       app: nfs-server
   spec:
     containers:
     - name: nfs-server
       image: gcr.io/google_containers/volume-nfs:0.8
       ports:
         - containerPort: 2049
         - containerPort: 20048
         - containerPort: 111
       securityContext:
```

```
          privileged: true
        volumeMounts:
          - mountPath: /exports
            name: gce-persistent-disk
      volumes:
        - name: gce-persistent-disk
          gcePersistentDisk:
            pdName: {{.Values.school.Id}}
            fsType: ext4
```

This nfs server is used by creating a persistent volume which refers to it as an nfs: using its cluster IP service which will also be created. Since persistent volumes are outside the scope of namespaces, the school Id which will be unique for every school is used for naming the pv. Here is how it might look like:-

```
apiVersion: v1
kind: PersistentVolume
metadata:
 name: {{.Values.school.Id}}
spec:
 capacity:
   storage: 10Gi #capacity of the nfs server
 accessModes:
   - ReadWriteMany
 nfs:
   server: #<service_name>.{{.Values.school.Id}}.svc.cluster.local
   path: "/"
```

Now gce pd can be consumed by simply creating a PersistentVolumeClaim. For this approach also a helm chart is already available from helm itself which can be tweaked to suit our needs.
https://github.com/helm/charts/tree/master/stable/nfs-server-provisioner

Approaches with a different database than mongoDB -

➔ If we replace mongoDB with a cloud database then the load on the cluster will reduce as we will not have to run mongo pods and it will simplify the architecture a lot but this approach may require significant rewriting of sugarizer-server code. The problem with rewriting code is that it will be time consuming to incorporate changes in new relases of sugarizer-server to sugarizer-server used here. Cloud Bigtable and Cloud Firestore and good options among cloud databases available on the Google Cloud Platform as

they provide scale insurance and they are also non-relational databases like mongodb making rewriting code to use them instead easier.
Here is a link comparing various cloud databases available on Google Cloud Platform -
https://cloud.google.com/products/databases

In my opinion MongoDB StatefulSet and gce persistent disk NFS approaches are good. I think the same database should be used on the original sugarizer-server and the one deployed on the Sugarizer School Portal so that upgrades could be easier. If we have to change the database then we should change it on the original version as well and go with either StateSet or gce pd nfs approach. However, this will be decided upon after discussion with mentors.

Timeline

Duration	Tasks
1st - 10th April	• Implement the front end of the Basic Demo application which will support basic features like requesting for a new deployment, destroying deployments and approving deployment requests. • Learn more about ansible and helm. • Understand the Sugarizer-Server codebase and work on making necessary changes to make it kubernetes ready.
10th - 20th April	• improvements to the demo application UI and discuss with the mentors how the actual UI might look like. • Explore the need of pub/sub and make necessary changes to the demo application. • Learn about Google Cloud DNS and Let's Encrypt. • Continue Working on Sugarizer Server kubernetes

	compatibility.
20th April - 4th May	• Learn more about various monitoring solutions like Prometheus, Prometheus-operator, grafana, influxdb etc. • Work on creating a kubernetes ready standalone image of the sugarizer server.
4th May - 1st June	**Community Bonding Period** • Explore the technologies in greater depth. • Communicate with mentors and other organization members to receive feedback on the work done so far, finalize UI, features and learn more about best practices to follow in a production environment. • Learn more about the project and how it will be implemented.
1st June - 8th June	• Implement suggested changes by mentors. • Complete the UI for the application except for the Monitoring section. • Complete Basic functionality like requesting for new deployments and approving deployment requests. • Write a Blog
8st June - 15th June	• Start Working on scaling functionality • Sugarizer Server made completely ready for kubernetes and a standalone image created. • Start working on routing through ingress and integration with cloud dns domain names • Write a Blog

15st June - 22nd June	• Scaling functionality Completed • Routing through Ingress Completed. • Add functionality to destroy deployments. • Add functionality to add more super admins. • Start Working on monitoring. • Write a Blog
22nd June - 29 June	• Finish The UI of the Super Admin Console • Add Basic functionality to School Console • Add functionality to add new admins in the School Console • Continue Working on monitoring. • Start Working on SSL, security encryption and lets encrypt • Write a Blog
29 June - 3 July	**July Phase 1 evaluation** **Progress:-** • A basic working version of the application will be created where the super admin will be able to login into the super admin console and create more super admin accounts. • The super admin will be able to approve deployment requests, destroy deployments • Scaling solutions implemented • Routing implemented • Domain Integrated • Write a Blog
3 July - 10 July	**Will not be able to work probably due to university exams will compensate for it by working more the following weeks**
10 July - 17 July	• Complete SSL and let's encrypt integration.

	• Integrate monitoring into the super admin console • Write a Blog
17 July - 27 July	• Completely Integrate Monitoring and resource administration by the super admin. • Finish the web interface - both the super admin console and the school console. • Write a Blog
27 July - 31 July	**Phase 2 evaluation** **Progress -** • Web interface Completely Implemented • All Proposed features developed and implemented • Complete Integration with Let's encrypt, SSL and Cloud DNS
31 July - 7 August	• Receive Feedback from the mentors and make requested changes. • Start working on the final ansible package to install the application onto GKE. • Write a Blog
7 August - 14 August	• Implement Feature requests received from the field. • Create an Image for the sugarizer school portal application. • Finalize the ansible package. • Write a Blog
14 August - 21 August	• Try to find and fix bugs • Thoroughly test the application for unexpected behaviour and make necessary changes. • Write a Blog

21 August - 24 August	• Prepare Documentation • Prepare for final evaluation • Write a Blog
24 August - 31 August	**Final Evaluation** • A complete end to end application developed • Ansible package developed for automating setup process which could create the required kubernetes infrastructure. • All features of the super admin console and school console functional.