Cloud Native Project Documentation

Group: **CloudJB** (Jaromir Charles, Benjamin Herrmann)

GitHub: https://github.com/JaromirCharles/PMS

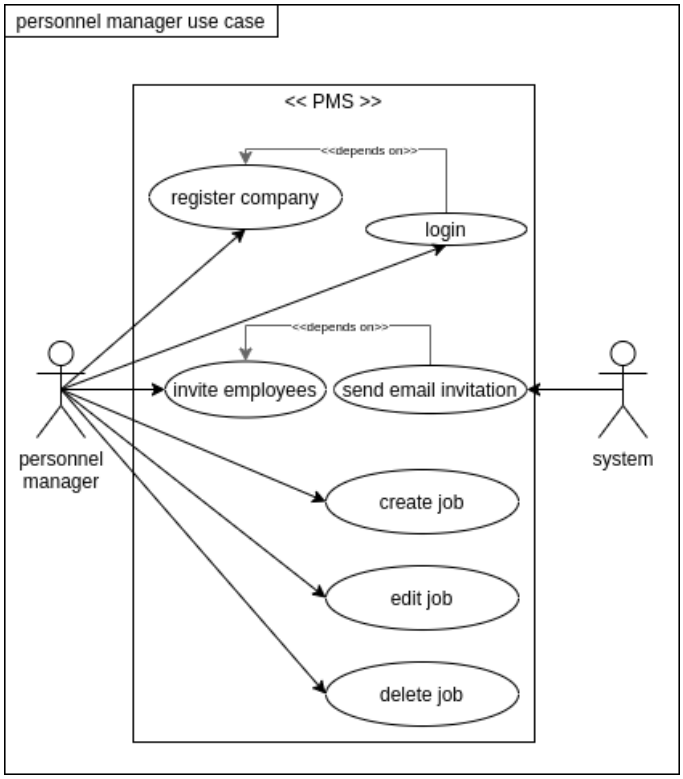
Table of Contents

Description

The main functionality of our multi-tenancy cloud native application is to allow:

- **tenants** (mainly the personnel managers) of a firm to organize their jobs and workers.
- **employees** to have a complete overview of their tenant's jobs.

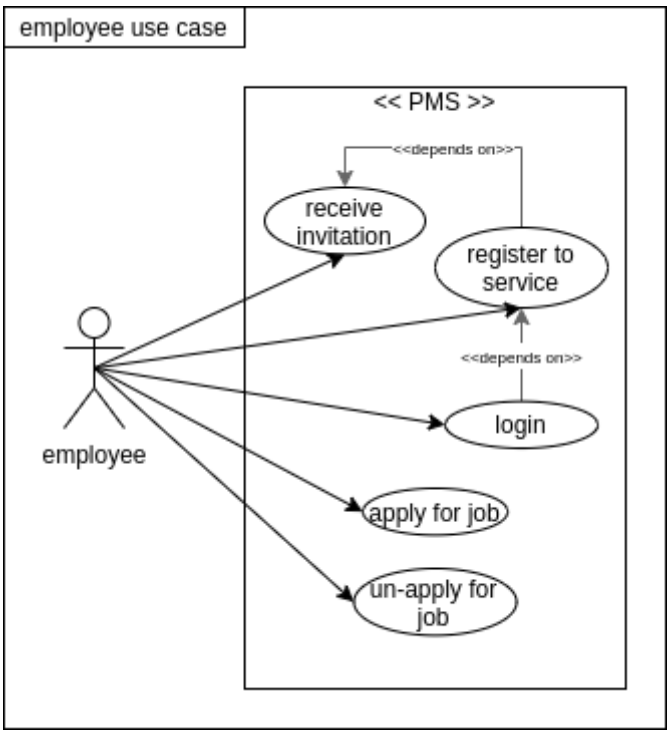
A better understanding of the main functionality of the application can be seen within the following use cases:



The diagram above depicts the main functionalities of a tenant. A tenant has the ability to register his company to use the `PMS` service. Upon successful registration he/she can log into the system and use its functionalities. A tenant has basically two important functionalities:

- **organizing employees** - the personnel manager can invite employees to use their `PMS` service whereby after submitting an employee's email address, an email invitation will be sent to that employee via our system.

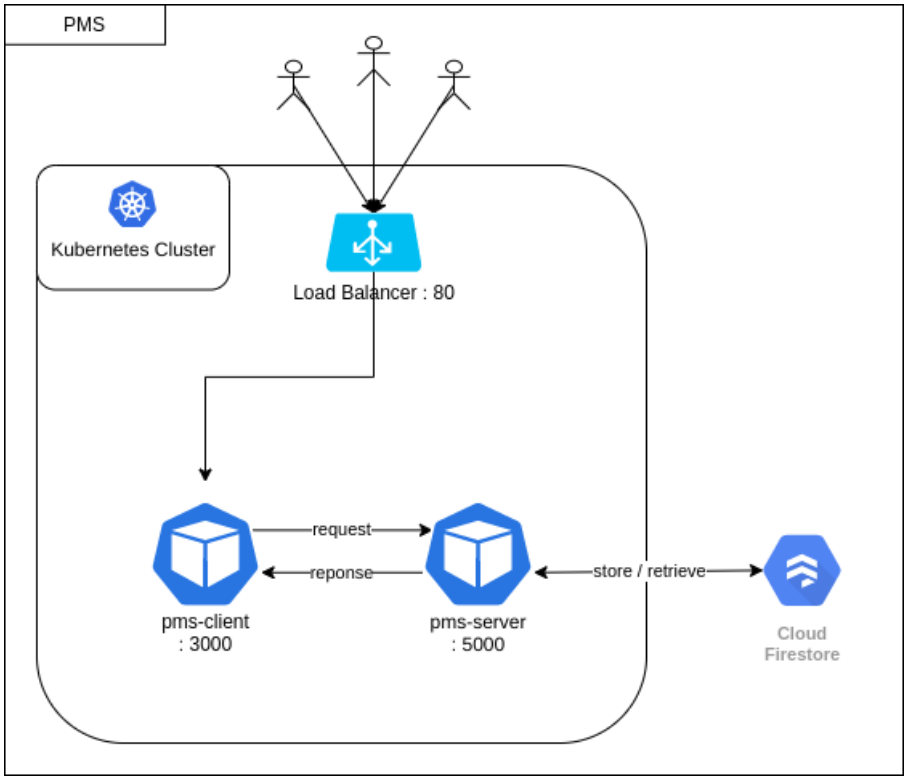
- **organizing jobs** - the other feature available to the tenant is to create, edit and delete jobs. While creating a job, fields such as the date, a job title and description, location, start time and number of needed workers for that job must be filled out. Employees can apply for these jobs, and the personnel manager can select from the provided list of applied workers who shall take part to accomplish said job.



The diagram above depicts the main functionalities of an employee. Upon retrieval of an email invitation, he/she can register themselves to use the `PMS` service of his/her company. After registration, when logged in, the employee is fist greeted with a list of the company's current available jobs. He/she can then with free will apply and un-apply for desired jobs. A respective view for the applied and upcoming (jobs the personnel manager selected the employee to part take in) jobs are also available.

Application Architecture & Design

Components and Interfaces



The image above depicts the components and interfaces of the `PMS` application. The application is broken down into two separate services: the `client` and the `server`, both being able to run independently from one another. The application runs within a Kubernetes Cluster.

The `client` (developed with react JS) runs separately in a pod. The client entails the User Interface with which the user interacts with and is visible on port `3000`. The `server` (a Node JS Express server) serves the frontend with its backend logic. Its REST API handles the requests to be made to the database (`Cloud Firestore`) creating, updating and retrieving desirable information. The server also runs in a pod within the cluster and is visible to other pods on port `5000`.

`Cloud Firestore` is a NoSQL document database allowing the `PMS` application to easily store and query data. The database has been made available via a secure service key so that the application (server side) has full access to perform desired operations on the database.

Within the cluster, the application is not reachable from the outside world. To expose the application outside of the cluster, a service of type `LoadBalancer` has been created to make the pods reachable via the Internet. The Load Balancer service has an external IP and redirects incoming traffic from port 80 to the application's frontend running internally on port 3000.

Important use cases

The most important use cases can be seen in the Description section. Further we'll look a bit more into the implementation behind these use cases.

Tenant: Register to PMS The tenant has the ability to register himself to use the `PMS` service. If the tenant clicks on the `REGISTER` button, he/she has the opportunity to fill out a form and submit it. He/she will now be able to log in with the email and password entered into the sign up form.

Tenant: Invite workers Inviting workers to use their `PMS` service is one of the main capabilities of the application. After logging in, the tenant can select the `Employees` menu and then has the ability to submit an email address, thereby requesting our system to send an email to that entered address. The tenant will also see a list of all the employees he/she has added in a table format.

Tenant: Create, Edit, Delete Job After logging in, the tenant has the ability to create, edit and delete a job with the click of a button. Clicking on `CREATE JOB` allows the tenant to fill out a form with job information like the date, title, description, etc. Submitting the form will result in the job being persisted to the database. The tenant can edit a job's information by simply clicking the `Edit` icon and clicking on `DELETE` will result in the deletion of the job from the system.

Tenant: View and accept applicants If the tenant selects a job, the job information as well as two lists: `selected` and `applied` will also appear where the tenant can see which employees applied for said job as well as assign employees to that job.

Above we discussed the main use cases of a tenant. Now we will look at the main use cases of an employee.

Employee: Register to PMS Upon receiving an email invitation to join a company's `PMS` service, the employee has the ability to submit a form with his log in information, thereby gaining access to the service.

Employee: Views After logging in, the employee can select between three different views from the menu bar. `Available jobs`, `Applied jobs` and `Upcoming jobs`. The Available jobs shows the current jobs that are available to part take in. The employee has the ability to apply for a job with a simple mouse click. In the Applied jobs view the employee can see all the jobs he applied for as well as to cancel the job. And within the final view, Upcoming jobs, the employee receives an overview of all the jobs he as been assigned to.

Cloud Provider Resources

The application is hosted on `Google Cloud` and uses a few resources that Google Cloud provides.

- **Google Kubernetes Engine**
    - Google's Kubernetes Engine has been used to run the application in a containerized environment. Kubernetes automates the deployment, scales and manages the `PMS` application. The advantages of using Kubernetes is that it does most of the work. It scales the application up and down and it handles the recreation of new pods if one happens to fail.
    - An alternative to GKE is Docker Swarm. Both have their advantages over each other, but GKE overcame Docker Swarm in important aspects such as cluster setup, scaling, logging, monitoring and load balancing.
- **Container Registry**
    - Google Cloud's Container Registry has been used to store, manage and secure the application's container images. These application images need to be available for Kubernetes, so that Kubernetes knows where it can find the images it needs to create pods.
    - Dockerhub is another well known central registry for storing `public` Docker images. We however want to control access to the application's images and thats why the Google's `private` Container Registry has been used.

- **Firebase Firestore**
  - The application uses a NoSQL Document store database because of its flexibility. It supports `agile software engineering` making implementing the application a lot easier and more flexible to sudden changes.
  - A NoSQL Document store database has been chosen over a standard SQL database because with a SQL database, the schema solubility would have been lost making the agile engineering much harder.
- **Compute Engines**
  - Google's Compute engines are being used indirectly, as the Kubernetes cluster creates virtual machines for its usage.

The five essential characteristics of a cloud service

`On-demand self-service`

The `PMS` application is an independent service. The users, both tenants and employees, have the ability to solely register themselves to use the application and perform the provided tasks the system offers. All requests are done automatically via the cloud infrastructure without any human interaction.

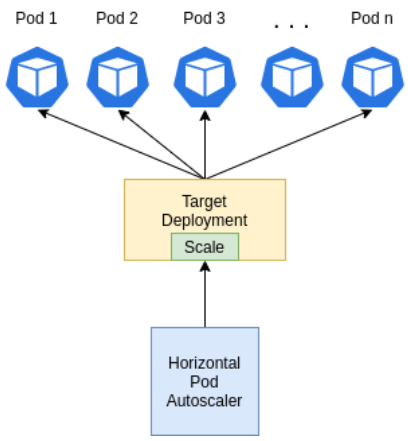
`Broad network access`

The Load balancer which was set up to expose the application to the outside world gives a public external IP address (`http://34.89.220.251/`) which allows all users to access our service at any time, from anywhere, from any device which is connected to the Internet.

`Resource pooling`

Resources are shared whereby the application is accessible to all users under the same IP address. Users share the same instance of the application which in turn is run on a definite number of Virtual Machines, thereby sharing computing resources. Storage resources are also pooled whereby all users share the same database instance. More about database resource sharing will be discussed in the Multi-user Multi-tenancy section.

`Rapid elasticity`

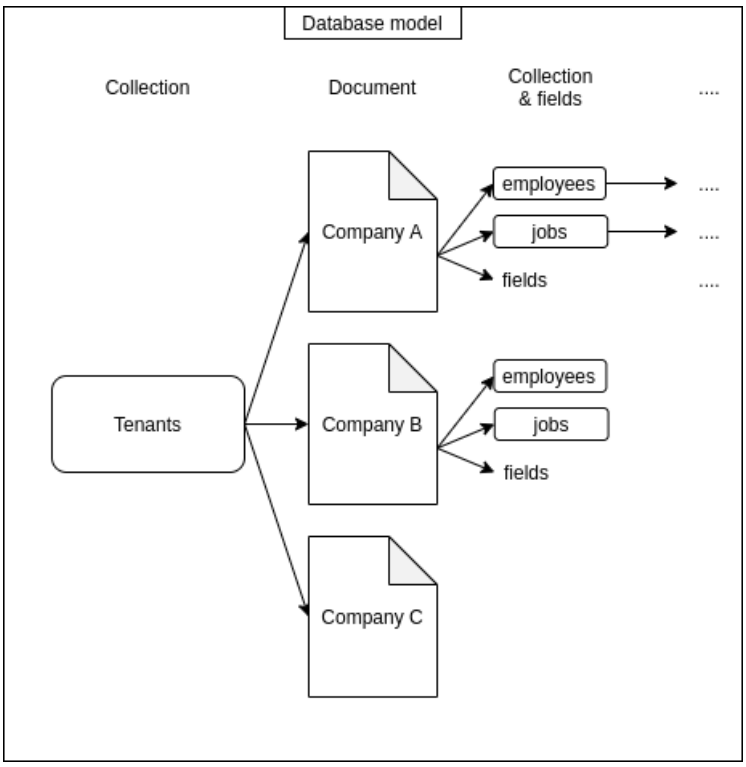


To be able to meet the demands of all users a `Horizontal Pod Autoscaler` was created which targets the client deployment, periodically adjusting the number of replicas of the scale target to match the average CPU utilization which was specified to `80%`. For current production, a maximum of `5` replicas and a minimum of `1` replica has been set. The number of replicas will scale up when the user demands increase and also automatically scale down if the resources are no longer utilized by the consumers.

Measured Service

One key feature of Google Cloud are the measured services. We just pay for what we consume. To optimize resource usage, we are able to automatically control the number of pods and also our computing resources as we described above in *Rapid elasticity*. Google Cloud monitoring provides us visibility into the performance, resource usage, computing resources, network usage, etc.

Multi-user Multi-tenancy



Different companies and users are using the same cloud provided instance of our application. Multi-user/tenancy is accomplished with our database model. With this approach we achieved resource sharing, but little isolation. One collection is used for all tenants, whereby each tenant in stored in a separate document, thereby isolating the tenants from one another. This is the first step in attaining tenant isolation. A sketch of the database model can be seen above.

Since all tenants share the same database resource, ensuring that no data is being leaked is a priority. Within the collection, each tenant has its own document, basically its own name-space per say. And within each tenant's document, the tenant's jobs and employees (both being collections) and other fields are stored, making this information only visible for said tenant.

Another step taken to ensure that no information is being wrongfully leaked to a tenant or an employee, is that the application ensures that each request made from a tenant entails the respective tenant's name as well as requests made by employees contain the employee's email together with his/her associated tenant's name.

Cloud Native with 12F

In the following we describe how the main characteristic of cloud native Containers, Continuous delivery, Micro-services and DevOps are implemented in our application.

With docker we have created two `containers` which provide an isolated workspace for our application. The client container contains everything related to the frontend and is reachable on port 3000. The server container is responsible for the backend logic, running on port 5000.

We support `Micro-services` with Containers and Kubernetes. Frontend and backend are separated from each other. They can be upgraded, scaled and deployed independently, see graph in section Application Architecture & Design. Further services can be easily integrated into the application.

Our `Continuous delivery` approach starts by making sure that our code is always in a deployable state. That means that changes on code level are tested locally and reviewed. The next step is to automatically deploy every build into a production-like environment. In our case this process of automatic deployment is not yet configured, so that we have to do it manually with three commands (`docker build`, `docker push` and `kubectl set image`).

Our `DevOps` approach is described in am upcoming section Operations.

Twelve Cloud Native factors:

(1) The application's `Codebase` is hosted with the version control system Git on GitHub. **(2)** For our dependency handling we use npm and yarn. All `Dependencies` are stored in the package.json file which is checked into version control to get started quickly in a repeatable way and it helps to easily track changes to dependencies. **(3)** As for now we have no `Configuration` external to the code which will be changed for different environments. **(4)** `Backing services` are not part of our application. **(5)** The `Build and release and run` phase is in our case the process to the next deployment on the server after committing a new feature or important changes (see Continuous delivery approach). **(6)** Our application is executed as one or more stateless `Processes`. Persisted data is directly stored in the Firestore database. **(7)** `Port binding` is used for our backend port 5000 and frontend port 3000. **(8)** `Concurrency` We configured Kubernetes to scale up or down the number of pods running in the cluster based on the CPU workload. **(9)** `Disposability` Our application can be gracefully started and stopped. **(10)** The application is designed to keep the gap between development and production small. `Dev/prod parity` The way from development/code commit to deploy is kept short and is therefore realizable in a few seconds. With Docker we can ensure that the application stack keeps its shape across environments. **(11)** For our cluster a logging agent from Google is automatically deployed on every node and collects `Logs` with relevant meta-data. They can be review using Cloud Logging. **(12)** Up to this point we have no `Admin processes` or repeatable administration tasks for our application.

Operations

Adding a new tenant

For demonstration purposes we added a registration page for the tenant where he enters his personal data (NB: In real life the entire registration process works obviously different). After successful registration his personal data will get saved in the firestore database. On database level a new tenant registration results in an new unique entry with the tenant's name in the `tenant` collection where all the other registered tenants are listed. Each tenant holds its custom data starting from this entry onwards. This helps us to separate the data on a tenant level. In addition, a new entry in the `login` collection with an encrypted password and role of user is added for login validation. Now the tenant is able to login with his personal data.

Installing the application on the cloud provider

Following we will discuss how the application has been installed to be run on our Cloud provider. These are steps that one can execute within a shell and requires `kubectl` to be installed, which is the Kubernetes command-line tool allowing one to run commands against Kubernetes Clusters.

1. **Set necessary parameters:**

```
export PROJECT_ID=apt-momentum-279610 # This is our Google application
project's ID
gcloud config set ${PROJECT_ID}
gcloud config set compute/zone europe-west3
gcloud auth configure-docker
```

2. **clone the application from GitHub**

```
git clone https://github.com/JaromirCharles/PMS.git
cd PMS
```

3. **Build and tag the Docker images**

   Our application entails Dockerfiles needed to build the Docker images (client and server) of the application.

```
cd server
sudo docker build -t gcr.io/${PROJECT_ID}/pms_server:v1 .
```

```
cd frontend
sudo docker build -t gcr.io/${PROJECT_ID}/pms_client:v1 .
```

4. **Push the Docker images to Google's Container Registry**

```
sudo docker push gcr.io/${PROJECT_ID}/pms_server:v1
sudo docker push gcr.io/${PROJECT_ID}/pms_client:v1
```

5. **Create a GKE cluster with a unique name**

   A GKE cluster has been created to run our `PMS` application.

```
gcloud container clusters create pms-cluster
```

6. **Deploy application to GKE**

   In this step we deploy the docker images created early to the GKE Cluster.

   ```
   kubectl create deployment pms_client --
   image=gcr.io/${PROJECT_ID}/pms_client:v1
   kubectl create deployment pms_server --
   image=gcr.io/${PROJECT_ID}/pms_server:v1
   ```

   We will also create a HorizonalPodAutoscaler resource which will scale the number of Pods between 1 and 5, based on CPU load.

   ```
   kubectl autoscale deployment pms_client --cpu-percent=80 --min=1 --max=5
   ```

7. **Expose the application to the Internet**

   While pods do have individually-assigned IP addressed, those IP addresses can only be reached from inside your cluster. Therefore we create a Kubernetes Service of type Load Balancer which will expose our application outside of the cluster. This type of Service spawns an External Load balancer IP for a set of Pods, reachable via the Internet.

   Use the kubectl expose command to generate a Kubernetes Service for the client and server deployment:

   ```
   kubectl expose deployment pms_client --name=pms_client_service --
   type=LoadBalancer --port 80 --target-port 3000
   kubectl expose deployment pms_server --name=pms_server_service --
   type=LoadBalancer --port 5000 --target-port 5000
   ```

The above steps can also be done via deployment and service yaml files. The following snippets shows how such files looks like.

```yaml
# client deployment yaml file
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pms-client-deploy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pms_client
  template:
    metadata:
      labels:
        app: pms_client
    spec:
      containers:
      - image: gcr.io/apt-momentum-279610/pms_client:v1
        name: pms_client
        imagePullPolicy: IfNotPresent
```

```yaml
# server service yaml file
apiVersion: v1
kind: Service
metadata:
  name: pms-server-service
spec:
  ports:
  - port: 5000
    targetPort: 5000
  selector:
    app: pms_server
  type: LoadBalancer
```

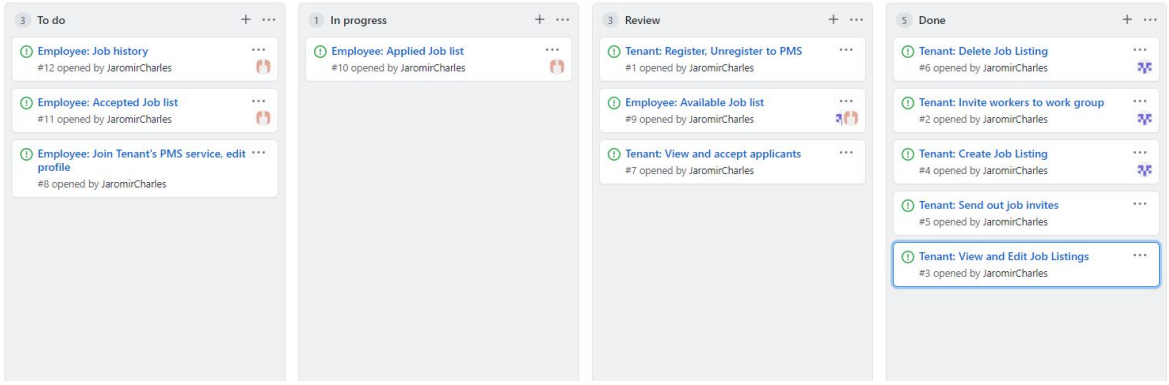
DevOps approach

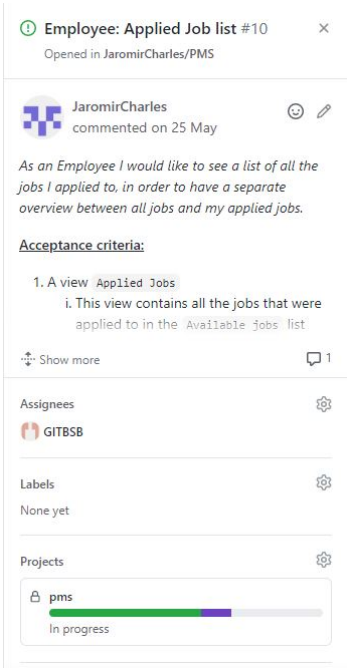
Since the DevOps approach is viewed as the collaboration between software developers and IT operations; Agile development, collaboration and communication are the keys of our DevOps approach.

For Agile development we tried to integrate Continuous Delivery practices which we already described in a previous section Application Architecture & Design. So we created an environment where building, testing and releasing software happens rapidly and frequently to constantly delivering high-quality software.

As a team of 2 developers, we utilized `GitHub` to manage the changes made to the source code as well as for collaboration. For communication we used `Slack` (chatting, sharing content) and `Discord` (talking and sharing knowledge).

To keep track of what needs to be done within the project, we used GitHub' Project Boards so that the team has an overview of what needs to be done, what is currently in progress, what needs to be reviewed and what has already been done.





Application's security model

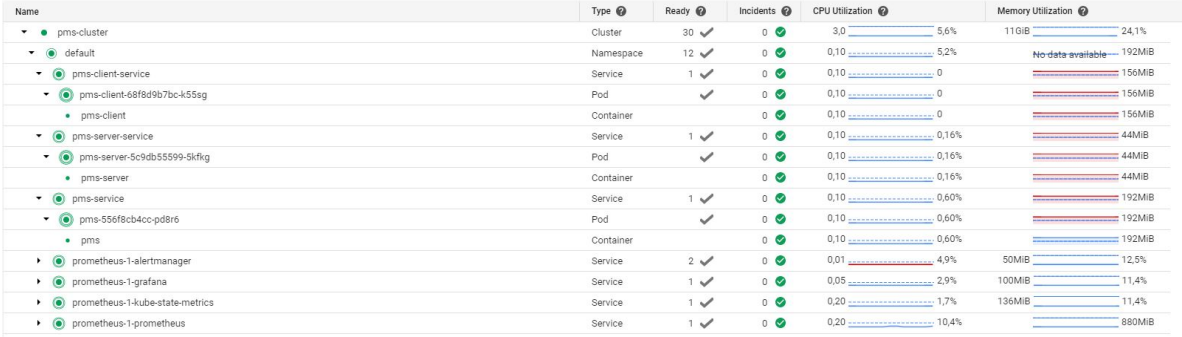
The most important aspect of our security model is the registration/login with a valid email and password. On database level passwords of users are stored encrypted with the help of bcrypt. Bcrypt is a Node.js-Module which incorporates not only a salt to protect against rainbow tables, moreover it remains resistant to brute-force attacks because it entails an increased iteration count which slows down the attack.

Another important aspect is `who` has `what` access to `which` resource. We have two groups defined, tenants (the companies) and users (the employees of the company). The login decides which role is applied and what the tenant/user will see and can do. The relevant data for logging in is stored in a separate table in Firestore including the role of the user. That means the application separates the data in the database on a tenant level as well as employee level. More information can be found in the section Multi-user Multi-tenancy

Our logging can also be mentioned here. All logs gets persisted in a data store from Google and can be reviewed using Cloud Logging. So we are able to monitor them to check user activities and take notice of conspicuous behavior.

Telemetry data

We installed Prometheus for better monitoring and health checks but due to the shortage of time, we were not able to configure it properly to our needs. So we were limited on gcloud monitoring (Dashboard Kubernetes Engine) where we could see some monitoring metrics of our Kubernetes nodes, pods and services. The given metrics which helped us to verify health and success of the application were CPU utilization, network usage and memory utilization. We are able to see any incidents and check the workload of the application at different time periods.

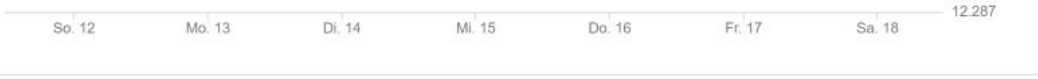


Pod Details: pms-client-68f8d9b7bc-k55sg

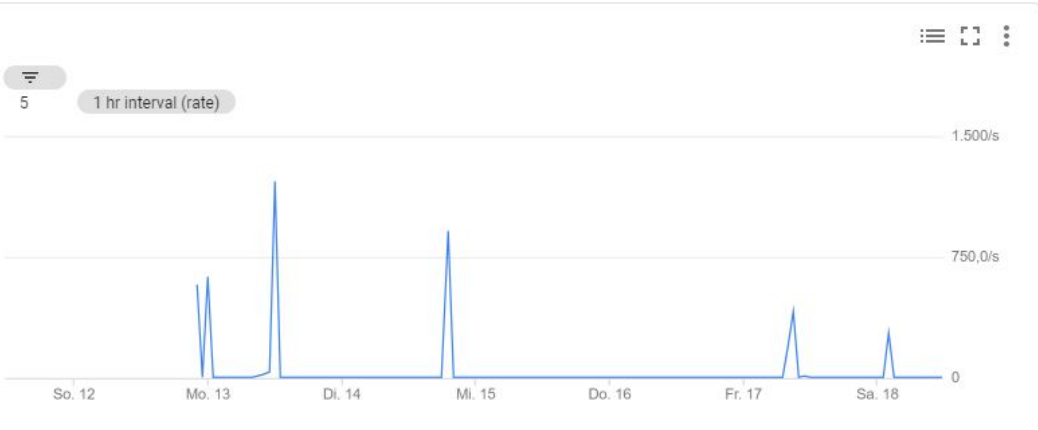
Ready Running MANAGE

0 Open Incidents

METRICS LOGS DETAILS



Network Usage



Cost Calculation & Charging Model

The running expenses of the application are the costs that we pay for the services being used from Google Cloud. The following table shows the resources being used and an almost precise estimate of their price.

Resource / Price	per hour in €	per month in €
Virtual Machine (3)	0.034	24.48 (73.44)
External IP	0.004	2.88
Firestore	-	-

The usage of Firestore resources is not billed on an hourly rate, but rather in the amount of storage used as well as the number of read, write, delete operations that are carried out. The price per GB of storage is 0.16€, 100,000 reads and writes cost 0.06€ and 0.18€ respectively. These numbers will fluctuate according to many different factors like, how many tenants are using the service, the number of employees each tenant has, the time of the year in which there are more jobs in certain months than other, etc. If we were to sum up the application's running cost, we will roughly be around 80€/Month.

To satisfy both large and small firms, we devised a charging model that we think is fair to both parties. Larger firms in turn will most likely have more employees and jobs (therefore the need to assign more workers to jobs per month) than smaller firms. With this in mind we created three packages for our tenants to choose from.

- **Small**
  - `1€` per user
  - `30€` per `250` assignments
- **Medium**
  - `1€` per user
  - `45€` per 500 assignments
- **Large**
  - `1€` per user
  - `60€` per 750 assignments

Within each package, `1€` is charged for each user. The difference by the packages lies by the number of assignments one has per month. An `assignment` can be seen has a job done by an employee. So if 5 people are needed for one job, this results to 5 assignments. If a company has the need for more monthly assignments, then another price plan can be made to suite that company's needs.