cloud native with 12F

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

With docker we have crated two `containers'` which provide 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 `microservices` with Containers and Kubernetes. Frontend and backend are separated from each other. They can be upgraded, scaled and deployed independently. (see graph in chapter "Application Architecture&Design) Further services can be easily integrated in 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. With Travis CI we ensure that all new code is automatically and consistently tested for errors. The next step is to automatically deploy every build into a production-like environment. In our case this process of automatically 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 chapter 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 app 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 app can be gracefully started and stopped. **(10)** The app 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 view seconds. With Docker we can ensure that the app stack keeps its shape and instrumentation across environments. **(11)** For our cluster a logging agent from google is automatically deployed on every node and collects `Logs` with relevant metadata. The can be review using Cloud Logging. **(12)** Up to this point we have no `Admin processes` or repeatable admin tasks for our application.

Operations

How a new tenant is added.

For demonstration purpose we added an registration page for the tenant where he enters his personal data (a short note: In real life this whole 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 tenants name in the 'tenant' collection where all the other registered tenants are listed. Each tenant holds its custom data starting from this entry on. This helps us to separates the data on a tenant level. In addition to that a new entry in the 'login' collection with the encrypted password and role of user is added for login validation. Now the tenant is able to login with his personal data.

How to install your application on the cloud provider

1. **Use Cloud shell with already installed gcloud und kubectl without any additional setup. Otherwise install manually.**

   Install the Google Cloud SDK. See

   Install the Kubernetes command-line tool. `kubectl` is used to communicate with Kubernetes.

   ```
   gcloud components install kubectl
   ```

2. **set defaults:**

   set project ID on the Console. our PMS ID is `apt-momentum-279610`

   ```
   export PROJECT_ID=apt-momentum-279610
   gcloud config set ${PROJECT_ID}
   gcloud config set compute/zone europe-west3
   gcloud auth configure-docker
   ```

3. **clone our app from git**

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

4. **Build and tag the Docker image**

   Our application comes with the Dockerfiles needed to build the Docker image of the application.

   container image backend:

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

   container image frontend:

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

5. **Test container image using local docker engine**

   ```
   sudo docker run --rm -p 3000:3000 gcr.io/${PROJECT_ID}/pms:v1
   ```

6. **Push the Docker image to Container Registry**

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

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

   Now that the Docker image is stored in Container Registry, you need to create a GKE cluster to run pms. A GKE cluster consists of a pool of Compue Engine VM instances running Kubernetes.

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

8. **Deploy app to GKE**

   Now we can deploy the docker image we built to the GKE cluster. Kubernetes represents applications as Pods, which are scalable units holding one or more containers. We will create a Kubernetes Deployment to run `pms` on the cluster. The deployment will have 3 replicas(pods). One deployment pod will contain only one container, the pms application docker image. We will also create a HorizonalPodAutoscaler resource that will scale the number of Pods from 3 to a number between 1 and 5, based on CPU load.

   Create a Kubernetes Deployment for the pms Docker image

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

   Set the baseline number of Deployment replicas to 3

   ```
   kubectl scale deployment pms --replicas=3`
   ```

   Create a HorizontalPodAutoscaler resource for your deployment

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

   To see the pods created, run the following command

   ```
   kubectl get pods
   ```

   * if it fails try FIX: `gcloud container clusters get-credentials pms-cluster --zone europe-west3-a`

9. **Expose the pms app to the internet**

   While pods do have individually-assigned Ip addressed, those IPs can only be reached from inside your cluster. Also, GKE Pods are designed to be ephemeral(lasting for a very short time), spinning up or down based on scaling needs. We need a way to `1)` group pods together into one static hostname, and `2)` expose a group of Pods outside the cluster, to the internet. Kubernetes Services solve for both these problems. Services group Pods into one static IP address, reachable from any Pod inside the cluster. GKE also assigns a DNS hostname to that static IP. To expose a Kubernetes Service outside the cluster, you will create a service of type *LoadBalancer*. 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 pms deployment:

```
kubectl expose deployment pms --name=pms-service --type=LoadBalancer --port
80 --target-port 3000
```

* Here, the `--port` flag specifies the port number configured on the Load Balancer, and the `-- target-port` flag specifies the port number that the `pms` app container is listening on.

10. **Deploy a new version of the pms app**

    One could upgrade the app to a new version by building and deploying a new Docker image to your GKE cluster. GKE's rolling update feature allows you to update your Deployments without downtime. During a rolling update, your GKE cluster will incrementally replace the existing `pms` Pods with Pods containing the Docker image for the new version. During the update, your load balancer service will route traffic only into available Pods.

    1. Build and tag a new `pms` Docker image.

    `docker build -t gcr.io/${PROJECT_ID}/pms-client:v2 .`

    NB: * when deploying frontend, change IP address in* *package.json* and in **mailer.js-> future .env file

    2. Push the image to Container Registry

    `docker push gcr.io/${PROJECT_ID}/pms-client:v2`

    3. Now one is ready to update the `app` Kubernetes Deployment to use a new Docker image

       Apply a rolling update to the existing deployment with an image update

       `kubectl set image deployment/pms-client pms-client=gcr.io/${PROJECT_ID}/pms-client:v2`

DevOps approach

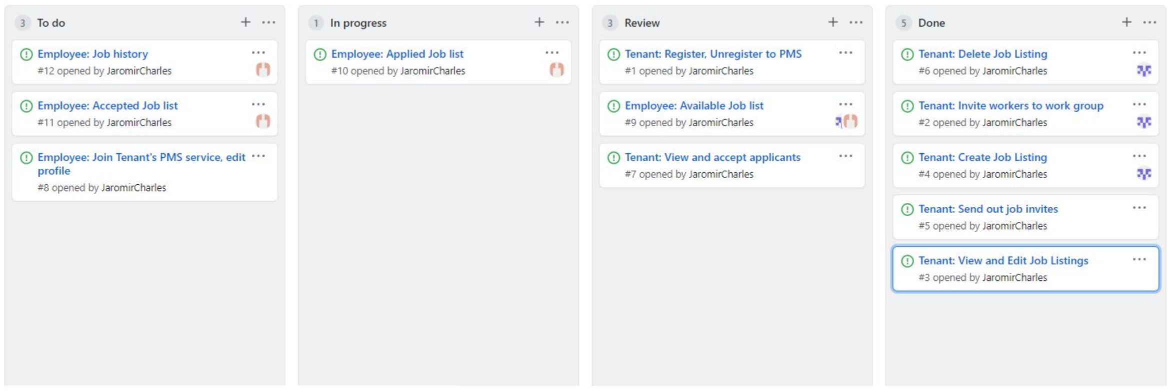
Since the DevOps approach is viewed as the collaboration between software developers and IT operations, Agile Development, collaboration and communication are the keys of our DevOps approach.

For Agile development we tried to integrate Continuous Delivery practices which we already described in the chapter *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). With the continuous integration platform Travis CI we ensured that all new code is automatically and consistently tested for errors.

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.

(C:\Users\BSB\Desktop\devOps2.JPG)



3 To do

- Employee: Job history
  #12 opened by JaromirCharles
- Employee: Accepted Job list
  #11 opened by JaromirCharles
- Employee: Join Tenant's PMS service, edit profile
  #8 opened by JaromirCharles

1 In progress

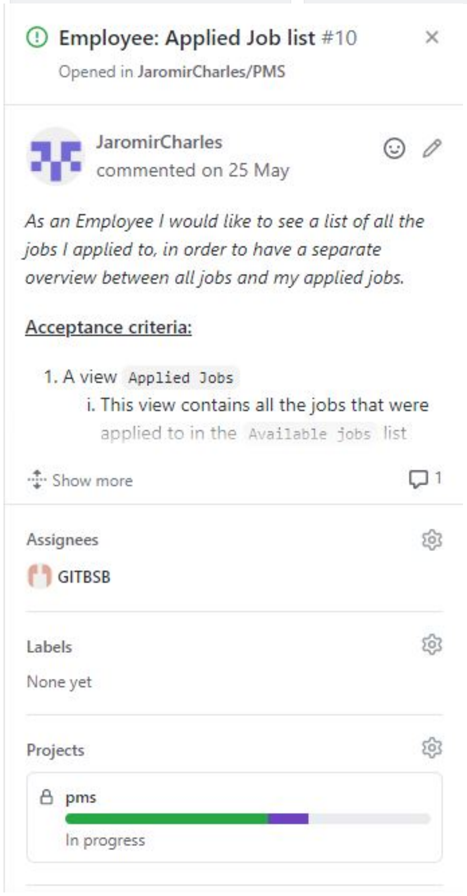
- Employee: Applied Job list
  #10 opened by JaromirCharles

3 Review

- Tenant: Register, Unregister to PMS
  #1 opened by JaromirCharles
- Employee: Available Job list
  #9 opened by JaromirCharles
- Tenant: View and accept applicants
  #7 opened by JaromirCharles

5 Done

- Tenant: Delete Job Listing
  #6 opened by JaromirCharles
- Tenant: Invite workers to work group
  #2 opened by JaromirCharles
- Tenant: Create Job Listing
  #4 opened by JaromirCharles
- Tenant: Send out job invites
  #5 opened by JaromirCharles
- Tenant: View and Edit Job Listings
  #3 opened by JaromirCharles



Employee: Applied Job list #10

Opened in JaromirCharles/PMS

JaromirCharles
commented on 25 May

As an Employee I would like to see a list of all the jobs I applied to, in order to have a separate overview between all jobs and my applied jobs.

Acceptance criteria:

1. A view `Applied Jobs`
   i. This view contains all the jobs that were applied to in the `Available jobs` list

Show more 1

Assignees
GITBSB

Labels
None yet

Projects
pms
In progress

Security model of your application

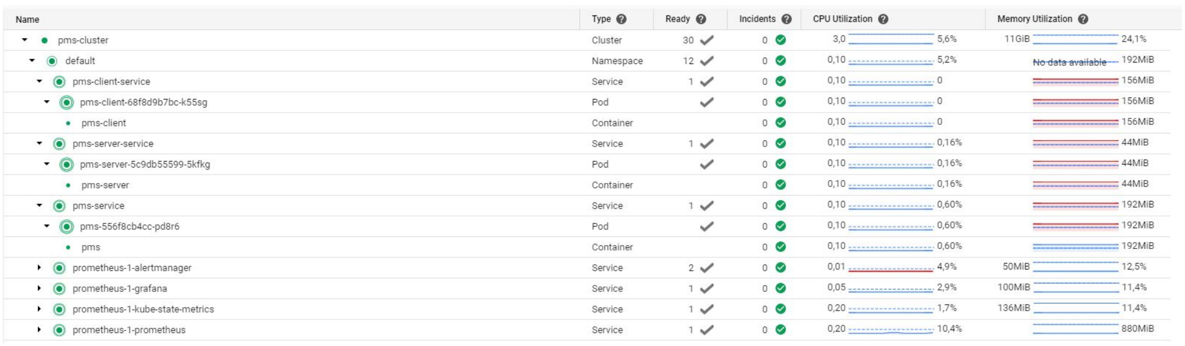
The most important aspect of our security model is the registration/login with a valid password. On database level passwords of users are stored encrypted with the help of bcrypt. Bcrypt is a Node.js-Module which incorporates not only a salt to protect against rainbow table, moreover it remains resistant to brute-force attack because an increased iteration count 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 login is stored in a separate table in Firestore including the role of the user. That means the application separates on database the data on a tenant level as well as employee level. More on chapter 'how multi-tenancy is implemented'.

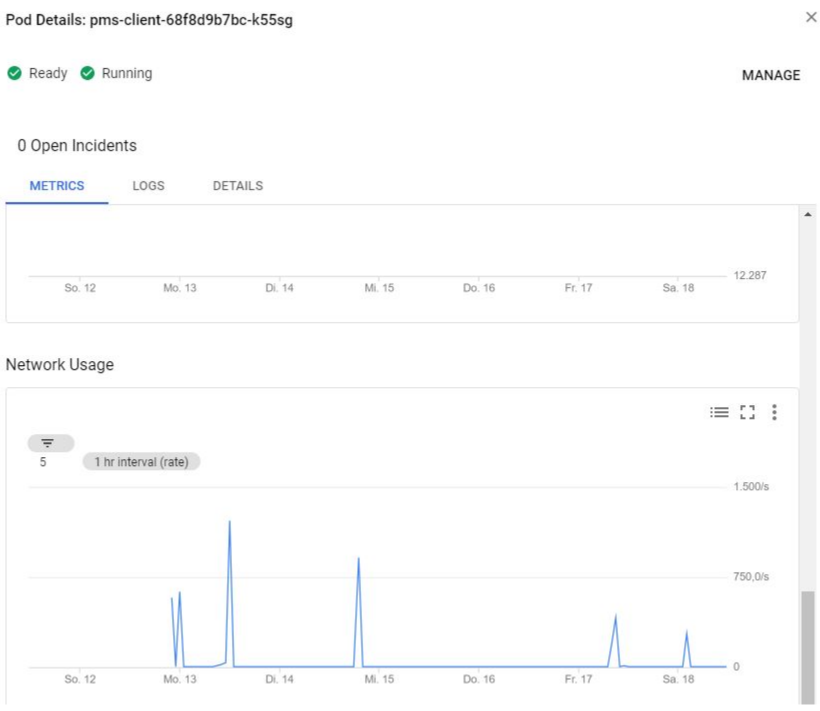
Our logging can also be mentioned here. All logs gets persisted in a data store from google and can be review using Cloud Logging. So we are able to monitor them to check user activities and take notice of conspicuous behavior.

Describe the telemetry data you collect for your application and how it can be used to monitor the health and/or success of the application.

We installed Prometheus for better monitoring and health checks but as other things were more important we didn't had the time to configure it properly to our needs. So we were limited on gcloud monitoring(Dashboard Kubernetes Engine) were we could see some monitoring metrics of our of 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. So we were able to see any incidents and check the workload of the application on different time periods.



Name	Type	Ready	Incidents	CPU Utilization		Memory Utilization	
pms-cluster	Cluster	30 ✓	0 ✓	3,0	5,6%	11GiB	24,1%
default	Namespace	12 ✓	0 ✓	0,10	5,2%	No data available	192MiB
pms-client-service	Service	1 ✓	0 ✓	0,10	0		156MiB
pms-client-68f8d9b7bc-k55sg	Pod	✓	0 ✓	0,10	0		156MiB
pms-client	Container		0 ✓	0,10	0		156MiB
pms-server-service	Service	1 ✓	0 ✓	0,10	0,16%		44MiB
pms-server-5c9db55599-5kfkg	Pod	✓	0 ✓	0,10	0,16%		44MiB
pms-server	Container		0 ✓	0,10	0,16%		44MiB
pms-service	Service	1 ✓	0 ✓	0,10	0,60%		192MiB
pms-556f8cb4cc-pd8r6	Pod	✓	0 ✓	0,10	0,60%		192MiB
pms	Container		0 ✓	0,10	0,60%		192MiB
prometheus-1-alertmanager	Service	2 ✓	0 ✓	0,01	4,9%	50MiB	12,5%
prometheus-1-grafana	Service	1 ✓	0 ✓	0,05	2,9%	100MiB	11,4%
prometheus-1-kube-state-metrics	Service	1 ✓	0 ✓	0,20	1,7%	136MiB	11,4%
prometheus-1-prometheus	Service	1 ✓	0 ✓	0,20	10,4%		880MiB



Pod Details: pms-client-68f8d9b7bc-k55sg

✓ Ready ✓ Running

0 Open Incidents

Network Usage