Virtualization and Cloud Computing

In this presentation we're going to talk only about the most challenging parts of the project, driving you through the problems we faced and the solutions we adopted to solve them.

Docker Registry

The first problem encountered when building the Docker Registry was the failure of the login task. This was caused by the fact that the two nodes tried to log into the registry when the container was not up and running yet and this resulted in an Internal Server Error.

So, in order to synchronize the task, we introduced a wait_for role to wait for the registry port to be open and ready to accept connections but, again, we faced the same problem. Later on, we discovered that our version of Docker had a terrible issue: It didn't close the ports after stopping containers and this made the wait_for role completely useless after the first run (also, we discovered that this problem was solved since Docker 10.12.2 but trying to installing this version gave us some problems due to the dependency with the docker-ce package). So, in order to solve this problem, we had to shut down the processes related to the registry manually.

We also included a retry condition for the login in order to try again after 10 seconds after the login fails. This is redundant when wait_for works but necessary when it doesn't.

Setup TLS

During the building of the registry we decided to instruct the nodes to trust it by using HTTPS protocol; this involved the generation of SSL certificates made up of different steps:

1. Creation of a private key.
2. Creation of a Certificate Sign Request (CSR), where all the information such as subject alt name, domain name etc… had to be specified.
3. Creation of a Self-Signed Certificate from the combination of the previously generated CSR and private key.

Then, we had to move that file into the CA directory of the manager and we had to invoke the *update-ca-certificates* executable in order to refresh the certification authorities trusted by the node. In addition, we had to restart the Docker service.

Furthermore, we automated the creation of the htpasswd file used by the registry to implement authentication functionalities: in fact, in the login task, both clients had to present the previously created certificates to be successfully authenticated.

Docker Registry cache

Also, we implemented a second registry which played the role of a pull-through cache; this means that when we need to pull an image we'll first look if the cache has it: if so, we won't need to download it, otherwise it will be pulled from the Docker Hub and will be kept in the storage for next

usages. To do this we had to add the registry-mirror specific in the Docker daemon file beyond modifying the configuration file.

PostgreSQL

We decided to deploy one replica of PostgreSQL containing two different databases: one for nextcloud and one for keycloak.

We leveraged the default postgreSQL entrypoint in order to create the databases, as provided in the examples; in this way each service has its own database.

The other choice would be more complex to implement and manage: for example, the default entrypoint would not be supported.

Keycloak

Another challenging part of the project was the building of Keycloak.

First of all, since Docker Stack does not have a form of synchronization between the services, we had to build our custom image of Keycloak (which is a little improvement of the one specified in the project rules) in order to make it wait for Postgres service to be up and running (otherwise, we wouldn't have been able to access the admin console). Also, to force the update of the image (entrypoint included) during the development we needed to erase and rebuild and push the image to the registry at each run.

In addition, we noticed that Keycloak allows to import an entire realm (which includes clients, roles and users) from all the JSON files placed in a specific folder so it was pretty easy to build a volume, map it to that folder and insert all the realm files in it. This saved us from various HTTP calls needed to create a realm with valid users.

Then, we discovered that, by default, Keycloak accepts connections and hostname resolution only using HTTPS protocol and this did not allow us to access the administration console. So, we had to disable this feature (--hostname-strict-https=false).

Furthermore, since it was placed behind a proxy, we had to enable the HTTP communication between them (flag –proxy=edge) and, from now on, reference it with the correct hostname (auth.localdomain) to enable automatic HTTP scheme, port and path resolution.

Nextcloud

As Keycloak, Nextcloud was also difficult.

First of all, it needed to wait for Keycloak to be installed and running (the operation could take some minutes) so we built a custom image (again, during the development phase we needed to force its update by removing and inserting the new image at each run).

To update the entrypoint we had to create one of our own (*set_config.sh*); this entrypoint:

- Waits for keycloak,
- Calls the installation of nextcloud
    - if it's not being installed by any other replica,
    - using the command: */entrypoint.sh apache2-foreground &*
      where *apache2-foreground* is the command needed to be run after the installation and *&* is used to put the job in background and continue the configuration.
- Configure OIDC.

During OIDC installation we had to add our domain name as a trusted domain, otherwise we wouldn't have been able to access the login form (connected with Keycloak).

Furthermore, when dealing with 2 replicas we needed to synchronize them. So, since only one of them must touch the configuration file and start the installation process (otherwise a modification applied by one instance could compromise the activity of the other), we implemented a mechanism based on locks: basically, the one of them that will acquire the lock will be the one responsible for the installation of Nextcloud and the configuration of OIDC, while the other will have just to start Apache.

Also, in order to keep sessions alive, we used StickySession labels to instruct Traefik to route the requests of a specific session to the same replica that served the first request for that session.

Fluent-bit

When dealing with the Fluent-bit configuration we faced the following challenges:

- *Parsers import*: in order to translate unstructured messages from Docker container into records a Docker parser was needed; in fact, its role was to take Docker logs using the tail input plugin and parse them into records.
- *Templating the file*: in order to verify that both nodes were sending logs correctly, we needed to distinguish them in some way; we used only one file, which was templated in a local folder in the two nodes with a label identifying the specific node.
- *Node exporter metrics usage*: as specified in the project rules we had to use the node_exporter_metrics input plugin, but we also had to allow the exporter to take more advanced system metrics from /proc and /sys directories. Moreover, we used the Prometheus_exporter output plugin to provide an endpoint from which data regarding the metrics could be fetched.

System metrics

In order to get system logs we leveraged the *systemd* input plugin redirecting the output to the Loki instance (in fact, Fluent has been also provided a URL used to push records into Loki) adding a different label and templating it, as explained before, basing upon the host.

Prometheus

Following the provided links, the first thing we noticed when dealing with Prometheus was its need to be allowed to access the Docker daemon running on a Swarm manager node in order to monitor its activity and get the metrics. So, this forced us to place this artifact on a manager node, map a volume to the Docker socket and specify the *root* user which has the permissions to scrape the daemon.

About the configuration, we chose to monitor the Swarm tasks keeping only the ones who are healthy and running and the ones having a *Prometheus-job* label (this fact was leveraged to apply relabeling). Also, since our services were forced to not expose ports, we had to instruct Prometheus about the endpoints where metrics were exposed: to do that we discovered that when monitoring Swarm tasks, it generates targets basing upon the __*address*__ label which has the following form *host:port*, where host and port were given in the configuration. So, in order to tell Prometheus where to get metrics we had to modify that label. Since we didn't have just a single target, cause we also had Cadvisor metrics to be scraped, we couldn't just set the target globally; instead, we leveraged regular expressions to apply relabeling for different targets instructing Prometheus about host and port to be scraped in order to get metrics.

Cadvisor

As specified talking about Fluent-bit node exporter metrics plugin configuration, since it provides an endpoint to scrape metrics from, we instructed Prometheus to fetch metrics from there. Basically, we had to do the same thing for Cadvisor on its default port 8080.

In addition, as said before, Prometheus has to be granted access to the Docker daemon so, again, we mapped a volume to solve this issue.

Grafana

During Grafana implementation, we faced some interesting challenges. The most relevant ones were:

- *Datasources provisioning*: we created a .yml file that listed all the datasources (in particular, Loki and Prometheus leveraging the Docker DNS that allows us to refer a service directly using its name) and mapped the folder in which it was contained to the Grafana datasource provisioning directory.
- *Dashboard provisioning*: same as before but the file created also referred the location of the JSON files representing the dashboards to be imported. Also, we automated the update of the JSON file of the provisioned dashboards from the UI by setting a specific flag.
- *Keycloak OIDC authentication*: in order to instruct Grafana to authenticate to Keycloak it has been provided the correct URLs to get the access token. Also, we derived the Grafana user role from the Keycloak service and allow Grafana to accept non TLS secured connections. In the end, we had to modify the hosts file of the container through compose to instruct him about where to find the authentication service and its domain name.