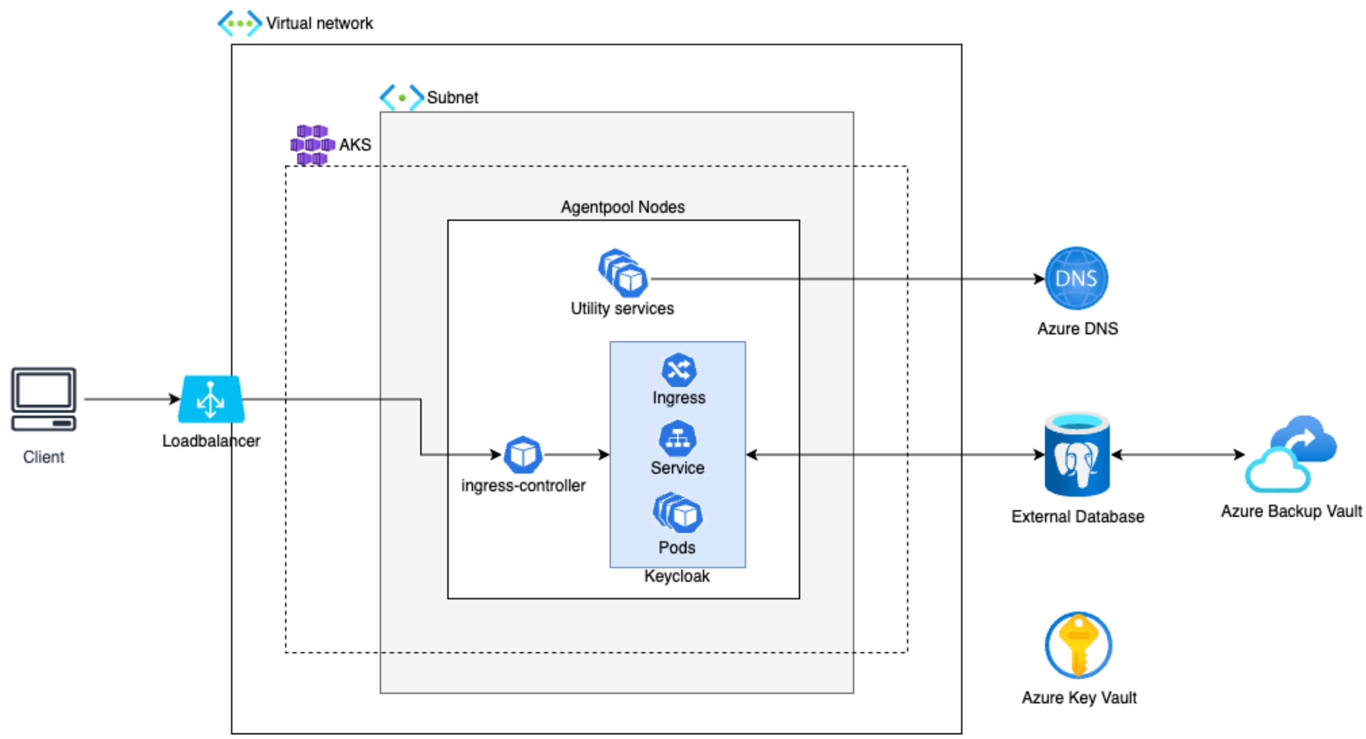
Implement Keycloak HA Setup with External DB

Repository

All code including this documentation is stored in a public git repository: https://github.com/Liquid-Reply/technical-challenge-wp3

Architecture

Current state



The architecture is based on Azure Kubernetes Service (AKS) where the compute nodes are spread across availability zones. This allows Keycloak to be easily deployed in a high availability setup. In this case three instances of Keycloak are deployed across three nodes, sharing the same external database.

Access to Keycloak is facilitated via an nginx ingress controller (running in AKS) with e2e https which is fronted by a load balancer. Let's Encrypt is used for certificates.

Azure Database for PostgreSQL Flexible Server is used as external database for Keycloak and access is facilitated with username/password for simplicity.

Azure Backup Vault is used for periodic backups of the database.

Azure DNS is used to resolve hostnames.

Azure Key Vault is used to store the DB admin password.

Further architecture improvements

> [!IMPORTANT] For a production grade setup at least the following aspects have to be considered and implemented from an infrastructure point of view:

- Have a production and development environment.
- Do not use default virtual network and subnets.
- Ensure traffic is private and use e.g. a bastion host to connect to the network.
- Use Azure NAT Gateway for internet access (ipv4).
- Use Azure Keyvault to store secrets and use Azure Private Link to connect to it from AKS.
- Use Azure Container Registry and a pull-through cache to store and retrieve OCI artifacts like container images or helm charts to reduce external dependencies. Connect to it via Azure Private Link.
- Ensure encryption at rest and encryption in transit. This includes managing encryption keys and certificates.

- Use AKS system/agent node pool only for critical workload. Use dedicated user node pools for further workload. Node pools should be deployed to all availability zones.
- Use e.g. Kubernetes RBAC with Azure RBAC integration to manage cluster access.
- Automate the deployment of infrastructure and workload to ensure consistency (use CI/CD pipelines and GitOps approach).
- Use Azure Private Link or VNet Integration to facilitate private access between AKS and Azure Database for PostgreSQL Flexible Server.
- Use managed identities to for authn/authz form Kubernetes workload to Azure services.
- Consider deploying Azure Database for PostgreSQL Flexible Server in `ZoneRedundant` mode and add read replicas which can be promoted to primary instances in case of failures.
- Consider deploying Keycloak in a multi-site setup depending on the criticality (see https://www.keycloak.org/high-availability/introduction).
- Consider using a web application firewall.
- Consider FinOps best practices to ensure reasonable cloud spending. This includes aspects like rightsizing and autoscaling of e.g. compute, storage and application resources.

Deployment

Infrastructure provisioning

The infrastructure components are provisioned via Terraform with its state saved in Azure Blob Storage. To provision the initial infrastructure needed for the terraform backend one can run `sh create-tf-backend.sh` from the root of the git repository. It is required to have Azure CLI (https://learn.microsoft.com/en-us/cli/azure/install-azure-cli) installed and to be logged in (`az login`).

Create a file called `postgresql_password` in `/infra/files/` and add a password. This will be written to Azure Key vault and used for the database deployment.

After that use `terraform init` to initialize your local environment. With `terraform plan` and `terraform apply` you can check which resources will be deployed and execute the deployment. You will be asked to provide a domain name which will be used to access Keycloak and the sample application. After all resources are successfully deployed you can continue with initializing Kubernetes.

Kubernetes initialization

Before deploying workload to Kubernetes some parameters must be set. The file parameters.yaml lists these parameters and where they must be set. After properly setting all parameters and after AKS is provisioned by Terraform make sure you can connect to the cluster as described in the Operations section and deploy utility services and Keycloak as lined out below.

> [!NOTE]
>
> In a more mature setup a GitOps tool would be provisioned which would then deploy and manage all required Kubernetes workloads.

Utility services

To deploy run the following commands:

1. `kubectl apply -R -f manifests/02_ingress-nginx/` (this deploys the nginx ingress controller)
2. `kubectl apply -R -f manifests/03_cert-manager/` (this deploys cert-manager to issue certificates)

Keycloak installation

To install Keycloak run the following commands:

1. `kubectl apply -R -f manifests/01_crds/` (this deploys required custom resource definitions for Keycloak)
2. `kubectl apply -R -f manifests/04_keycloak/` (this deploys Keycloak)

This will deploy Keycloak with three instances in a HA setup (see `spec.instances` in /manifests/04_keycloak/07_keycloak.yml).

To check that the Keycloak instance has been provisioned in the cluster run:

`kubectl get keycloaks/keycloak -n keycloak-system -o go-template='{{range .status.conditions}}CONDITION: {{.type}}{{"\n"}} STATUS: {{.status}}{{"\n"}} MESSAGE: {{.message}}{{"\n"}}{{end}}'`

The output for the successful deployment should look like this:

```
CONDITION: Ready
  STATUS: true
  MESSAGE:
CONDITION: HasErrors
  STATUS: false
  MESSAGE:
CONDITION: RollingUpdate
  STATUS: false
  MESSAGE:
```

After successful installation Keycloak is available via https://host.keycloak.<your_domain>/.

Service integration

We are using Grafana as sample application to test the Keycloak integration. The integration of Keycloak in a tool depends on the configuration parameters the tool offers. In case of Grafana it is possible to integrate Keycloak via the `GF_AUTH_GENERIC_OAUTH_*` environment variables which can be set e.g. in the Deployment resource. The values of `GF_AUTH_GENERIC_OAUTH_CLIENT_ID` and `GF_AUTH_GENERIC_OAUTH_CLIENT_SECRET` are provided via Kubernetes Secrets which are populated with values obtained from Keycloak when creating a client in a realm.

To rollout the sample application run `kubectl apply -R -f manifests/05_sample-application/`.

After successful rollout Grafana will be available via https://grafana.keycloak.<your_domain> and you can use OAuth login for users defined in Keycloak.

Operations

The following sections include commands to be executed in the CLI. To ensure that these commands are successful please check the prerequisites:

- Azure CLI (https://learn.microsoft.com/en-us/cli/azure/install-azure-cli), kubectl (https://kubernetes.io/docs/tasks/tools/) and kubelogin (https://azure.github.io/kubelogin/install.html) are installed (https://learn.microsoft.com/en-us/cli/azure/install-azure-cli) on the machine that has access to the cluster. This can be a local machine if cluster has public access enabled (which is the case for the current infrastrucutre setup) or e.g. a bastion host if the cluster is private (should be private for production). To connect to the bastion host you can use the Azure portal (https://portal.azure.com/) to identify the VM and check the connection possibilities.

Maintenance

Connect to cluster

To connect run the following commands (you can also find them in the Azure Portal via the "Connect" option in the overview tab of the respective Kubernetes cluster):

1. Login on the cli: `az login` (use the flag `--tenant <tenant_id>` if you have access to multiple tenants)
2. Set subscription where the cluster is deployed: `az account set --subscription <subscription_id>`
3. Retrieve access credentials: `az aks get-credentials --resource-group <cluster_resource_group> --name <cluster_name> --overwrite-existing`
4. Use kubelogin plugin for authentication: `kubelogin convert-kubeconfig -l azurecli`
5. Now you can use commands like `kubectl get nodes` to interact with the Kubernetes cluster.

Backup & Restore

Kubernetes resources

All Kubernetes manifests are stored in Git. The commit history is the backup which can be used to restore certain manifest version which can then be deployed.

Azure resources

The database is the most crucial resource for Keycloak in a Kubernetes based environment as multiple Keycloak instances can be created/deleted with ease, however they all rely on a production ready database that can is performant and reliable. It should be periodically backed up and be available to be quickly restored in case of a disaster.

The Azure Database for PostgreSQL flexible server can be backed up in multiple ways. The restore steps depend on the type of backup that is used to restore the database. The following backup options are supported

1. Automated daily backups as part of the Azure Database for PostgreSQL flexible server deployment (supports point in time restores only)
2. Configurable periodic backups as part of a Backup Policy created in an Azure Backup Vault (supports point in time restores and backup on demand)

3. On demand export/import of database using PostgreSQL tools pg_dump and pg_restore/psql.

In case of Automated daily backups, restoring a database involves restoring from a snapshot. This creates a new Azure Database for PostgreSQL flexible server and the Keycloak deployment is simply pointed to the new server or DNS can be used to point to the new server.

In case of backups stored in an Azure Backup Vault, the restore process involves first restoring a specific backup from the Azure Backup Vault to an Azure Storage Account Container and then using PostgreSQL tools pg_restore/psql to restore the database.

The database at any point in time can be backed up and restored using PostgreSQL tools pg_dump and pg_restore/psql.

```
# To backup the contents of a database
# -W - prompts for password input
# -U - server username (admin database username)
# -d - database name
# -F, --format=c|d|t|p output file format (custom, directory, tar, plain
text (default))

pg_dump -W -h $DB_SERVER_HOSTNAME -U $DB_SERVER_USERNAME -Ft -d $DB_NAME >
keycloak-db.tar

# To restore the contents of a database
# -W - prompts for password input
# -U - server username (admin database username)
# -d - database name
# -F - --format=c|d|t backup file format (should be automatic)
pg_restore -W -h $DB_SERVER_HOSTNAME -U $DB_SERVER_USERNAME -c -d $DB_NAME
< keycloak-db.tar
```

Scaling

Depending on the load on the system different parts can be scaled. Check the following.

Kubernetes

Either the number of nodes or the instance type can be scaled in /infra/main.tf. To scale the number of nodes in a node pool adjust the `node_count` property. The instance type can be changed with the `vm_size` property.

Database

Database settings can be adjusted via the `sku_name`, `storage_mb` and `storage_tier` parameters in /infra/database.tf.

Keycloak

Scaling Keycloak is equivalent to adjusting the `spec.instances` configuration parameter in manifests/04_keycloak/07_keycloak.yml. To rollout configuration changes see Change Keycloak configuration.

Change Keycloak configuration

> [!NOTE]
>
> Configuration changes should first be tested in a development environment before being rolled out to production to avoid unwanted downtime.

To change configuration parameters of Keycloak adjust the corresponding values in manifests/04_keycloak/07_keycloak.yml and rollout the changes via:

`kubectl apply -f manifests/04_keycloak/07_keycloak.yml`

(to check available parameters see https://www.keycloak.org/operator/advanced-configuration & https://www.keycloak.org/server/all-config)

As for the Keycloak installation you can check that the Keycloak instance has been provisioned in the cluster by running:

`kubectl get keycloaks/keycloak -n keycloak-system -o go-template='{{range .status.conditions}}CONDITION: {{.type}}{{"\n"}} STATUS: {{.status}}{{"\n"}} MESSAGE: {{.message}}{{"\n"}}{{end}}'`

The output for the successful deployment should look like this:

```
CONDITION: Ready
  STATUS: true
  MESSAGE:
CONDITION: HasErrors
  STATUS: false
  MESSAGE:
CONDITION: RollingUpdate
  STATUS: false
  MESSAGE:
```

If configuration changes are performed during a maintenance window Keycloak can also first be scaled down to zero (equivalent of stopping Keycloak) with:

```
# Stop the Keycloak instances
kubectl --namespace keycloak-system patch keycloak keycloak  --type=json --patch='[{"op":"replace","path":"/spec/instances","value":0}]'

# Perform maintenance tasks
```

Running `kubectl apply -f manifests/04_keycloak/07_keycloak.yml` as mentioned above will scale up the instances again (equivalent of starting Keycloak again).

Update Keycloak version

Keycloak is deployed via an operator. To upgrade to a new version check the releases in GitHub (https://github.com/keycloak/keycloak/releases) and read the corresponding official upgrade guide (https://www.keycloak.org/docs/latest/upgrading/index.html) to assess if configuration changes to Keycloak are required as well. The new Keycloak version can be specified in the Kubernetes Keycloak manifests in manifests/04_keycloak (especially the image tag in the `Deployment` resource). Make sure to also set the correct version in all labels/annotations to keep consistency and additionally update the Custom resource Definitions if required (manifests/01_crds/).

Version upgrades and configuration changes should be tested in a development environment before being rolled out to production. In case zero downtime deployments are not possible a maintenance window should be communicated.

After setting all required values roll out the changes as described in Keycloak installation.

Troubleshooting

> [!NOTE]
>
> To use the following kubectl commands prerequisite is that you are able to connect to the cluster (see here).
>
> In the kubectl commands you can also use `get -o yaml` instead of `describe` to get the configuration details of the individual Kubernetes resources.

Check application logs

You can directly access Keycloak application logs from Kubernetes.

To see all Keycloak logs: `kubectl logs sts/keycloak -n keycloak-system`

To see specific pod logs: `kubectl logs pod/<pod_name> -n <pod_namespace>`

Check Node health

- List nodes and their status: `kubectl get nodes`
- Describe Nodes that are not healthy and check `Events`: `kubectl describe node <node_name>`

Check Pod health

- List all Pods and check `Status`: `kubectl get pods -A`
- List all Keycloak Pods and check `Status`: `kubectl get pods -n keycloak-system`
- Describe Pods that are not healthy and check `Events`: `kubectl describe pod <pod_name> -n keycloak-system` (example is for Keycloak)
- Check logs of Pods that are not healthy

Check network path

You can use a bottom-up approach to identify issues along the network path.

- Check if Keycloak Pods are healthy (see here) and fix potential issues
- Check if Keycloak Service points to backend Pods:
  - `kubectl describe svc keycloak-service -n keycloak-system`
  - `Endpoints` should list the IPs and ports of the Keycloak Pods. If that is not the case then the Service is probably misconfigured (e.g. wrong selector or targetPorts).
- Check if Keycloak Ingress points to Service:
  - `kubectl describe ing keycloak-ingress -n keycloak-system`
  - backend should point to the keycloak service
- Check if ingress controller is up and running: Follow the steps to check pod health (see here) but use `ingress-nginx` instead of `keycloak-system`
- If the previous steps didn't reveal issues you should check the load balancer in Azure, especially if the health probes are successful and if the kubernetes nodes are listed in the backend pool. Additionally you should also check if the DNS records are correctly configured.

Check overall cluster health

The following command will print very detailed information about the overall health of the cluster:

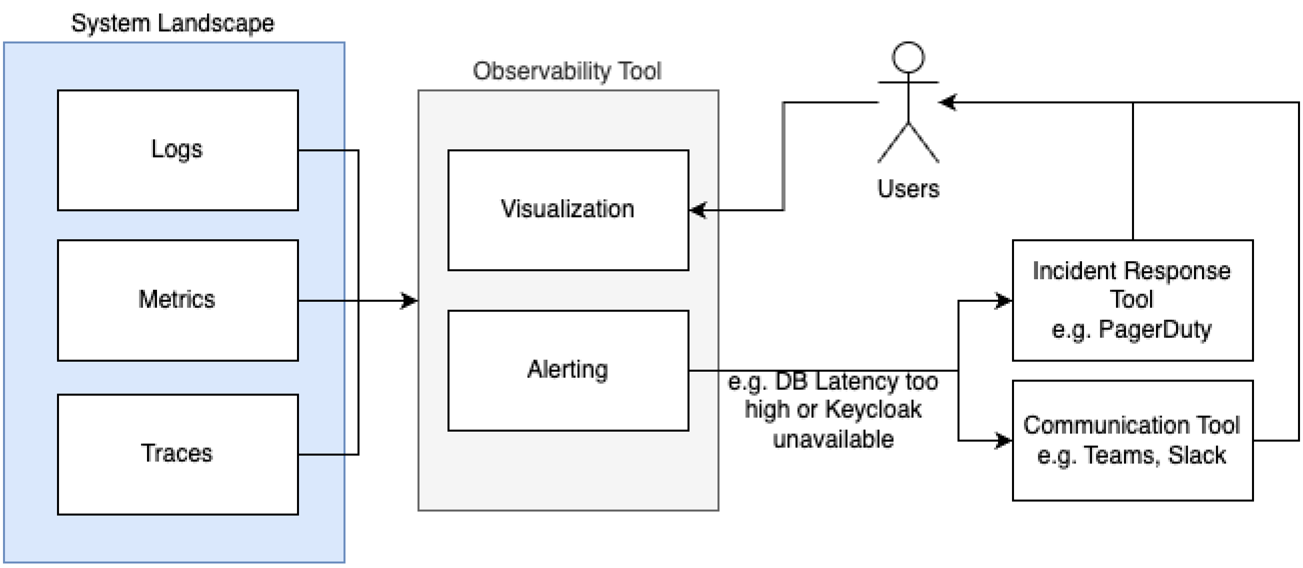
`kubectl cluster-info dump`

You can use the output to search for errors.

Check Database Health

Use a monitoring system or the Azure Portal to check if the database used by Keycloak is healthy. Take a look at the server metrics (e.g. failed connections, alive status) and if the database used by Keycloak exists.

Monitoring



A proper monitoring solution includes an observability tool that consumes different signals like logs, metrics and traces from the whole system landscape (infrastructure and applications). This data needs to be visualized to quickly understand it and an alerting system needs to established which notifies incident response tools and notifies teams.

Logs

The observability system should collect logs from Kubernetes workload including Keycloak and the utility services. To not consume too many logs (and therefore generate noise and unwanted costs) it is sufficient to focus on logs with a log level of warning and above. This is usually supported by log collectors. For access logs from the ingress controller this needs to be evaluated if `info` logs should also be consumed.

Besides logs from Kubernetes it is also important to check infrastructure component logs e.g. from Azure Database for PostgreSQL Flexible Server and other system that might produce logs.

Audit logs should also be consumed to track potential unwanted configuration changes or access.

Metrics

Metrics from infrastructure and application resources should be monitored and proper alerting established. At least these metrics should be covered:

- CPU
- Memory
- Disk pressure
- DB connections
- DB latency
- Storage consumption
- Kubernetes Pod status
- Kuberentes Pod restarts
- Kubernetes Node status
- Keycloak usage metrics (e.g. login errors, registrations)

Traces

To get an end-to-end view on system performance a tracing solution should be established. This helps to quickly identify bottlenecks and points of failure. This can also help in scaling and right-sizing decisions.

Alerting

Based on the logs, metrics and tracing data a proper alerting should be established that allows for quick discovery of unexpected behavior. Depending on the priority of alerts incidents can be triggered in downstream tools like PagerDuty and/or communicated to responsible teams via Teams, Slack or alike.