enaio cloud in AKS

Ziel des Termins am 18.02.2019, war es die enaio cloud Komponenten in einem Kubernetes Cluster in Azure zu deployen.

Rückblick

Im ersten Termin wurden im namespace "infrastructure" die Komponenten:

Komponente	enaio Bezug
elasticsearch	Suche, Rechteprüfung, Volltext (Ergbenis Contenanalyzer) Speicher, alle Anfragen werden initial ans elasticsearch gestellt
keycloak	Identity Provider, Authentifizierung, Nutzer Verwaltung, Zuordnung Nuzer <-> Rolle
gitea	git im Cluster, in dem die enaio cloud config Dateien gespeichert werden
minio	Proxy für Blobstore, gespeichert wird in ein Azure Blobstore, für den Nutzer von minio gibt es eine einheitliche S3 Schnittstelle
redis	In Memory Speicher, wird zum kurzeitigen Sperren von Objekten beim Update verwendet
cockroach	Datenbank, Ort der Wahrheit, 3 Tabellen enaio, 2 Tabellen liquibase (Java Framework für DB Abstraktion)
activemq	Messaging System, enaio cloud verewendet amqp 1.0 Version

installiert.

Im Anschluß wurden ein Ingress Controller und ein certmanager installiert.

Der Keycloak wurde über einen Ingress nach außen freigegeben und ist unter der Adresse `https://auth.redline-cloud.com` erreichbar.

Einrichten eines Realms in Keycloak

Im ersten Schritt wurde im Keycloak per web-admin unter https://auth.redline-cloud.com ein realm `hackathon` erstellt. Dann wurde in dem realm `hackathon` ein Nutzer `root` erstellt.

In einem CI System könnte dieser Schritt automatisiert werden. Realms können im JSON Format exportiert und importiert werden. Weiterhin stellt Keycloak eine umfangreiche REST-API bereit `https://www.keycloak.org/docs-api/4.5/rest-api/`.

gitea Konfigurieren

Im ersten Termin wurde gitea im Cluster installiert. Dieser Dienst soll als git repository im Cluster vom enaio configservice verwendet werden.

Als Datenbank in gitea wurde sqlite ausgewählt.

Im Anschluß wurde ein Nutzer `enaio` erstellt. Mit dem Nutzer `enaio` wurde ein leeres privates git repository `enaio-config` erstellt.

Nach dem deployen des Pods muss gitea einmalig initial konfiguriert werden. Um die Web Oberfläche von gitea zu erreichen wurde der kubectl Befehl port-forward benutzt: `kubectl port-forward -n infrastructure gitea-deployment-9f7bbcb7d-7shm7 3000`

Dieser Befehl lässt den kubectl Client als proxy zwischen der lokalen Maschiene und dem Container im Pod agieren. Alle Rest Requests unter http://127.0.0.1:3000 werden an den Container auf Port 3000 weitergeleitet.

Secrets im Cluster angelegt

In Kubernetes gibt es das Konzept von secrets um sensible Daten zu speichern und zu verwalten. https://kubernetes.io/docs/concepts/configuration/secret/

Secrets enthalten unter `data:` key value Paare, wobei der Wert base64 enkodiert ist. Für die Zugangsdaten zu `minio`, den im vorigen Schritt angelegten gitea Nutzer `giteauser` und `cockroach` wurden im namespace `infrastructure` secrets erstellt. Dazu wurden die Werte base64 encodiert mit `echo -n "Wert" | base64` und dann in eine yaml Datei gespeichert. Als Vorlage wurde das bereits vorhandende secret für keycloak `keycloak-http` verewendet. (abrufen mit `kubectl get secret -n infrastructure keycloak-http -o yaml > keycloaksecret.yaml`)

Beispiel cockroach: `kubectl apply -f cockroachsecret.yaml -n infrastructure`

Anzeigen der secrets mit: `kubectl get secret -n infrastructure`

Erstellen der Datenbank in cockroach

Für das Erstellen der Datenbank wurde ein Kubernetes Job verwendet.

```
kubectl apply -f cockroach-enaio-init.yml
```

Der Job führt die folgenden SQL Befehle in cockroach aus:

- CREATE DATABASE IF NOT EXISTS enaio;
- CREATE USER enaio;
- GRANT ALL ON DATABASE enaio TO enaio;

Ein Job in Kubernetes führt einen oder mehrere Pods aus bis diese sich erfolgreich beendet haben. Im Gegensatz dazu werden Pods in deployments und statefulsets immer wieder gestartet, bis die konfigurierte Anzahl an Pods im Cluster läuft.

enaio cloud Komponenten per gitlabci

Ziel ist es die enaio Komponenten mit gitlab automatisch zu deployen.

enaio cloud Komponenten

Das Produkt enaio cloud 1.0 umfasst die Services unter `enaio infra` und `dms core`. Im Cluster werden zusätzlich der client und die Komponenten für die asynchrone Textextraktion installiert.

enaio service	Funktion
`enaio infra`	
configservice	jeder enaio service versucht nach dem Start den configservice aufzurufen und seine im deployment angegbenen config Datein zu laden, jeder enaio service braucht den configservice
discovery	wie in blue und redline, jeder enaio service versucht sich an der Eureka registry anzumelden, enaio services finden sich über die discovery
admin	wie in blue und redline, Ausnahme swagger-ui Links im Admin zeigen auf die internen IP -> nicht mit dem Browser aufrufbar
`dms core`	
api-gateway	stellt die offizielle enaio cloud api bereit, zentraler Einstiegspunkt im System, alle Anfragen sollen über das api-gateway gehen
autentication	Schnittstelle zur Außenwelt, prüft und erstellt jwt Token für die Nutzer
audit	für jeden Nutzer Zugriff wird ein Audit Eintrag geschrieben, Audit service liest, schreibt und löscht Einträge in der Audit Tabelle in der Datenbank
contentanalyzer	versucht Text aus Dokumenten wie pdf, office, emails etc. zu extrahieren
registry	liest, schreibt und löscht Metadaten in die Datenbank, validiert die Metadaten anhand des Schemas
repository	liest, schreibt und löscht die Binär Daten, unterstützt nativ s3 und zip Ablage (unter einem Konfigurierten Pfad werden die Datein gezippt abgelegt)
organization	verwaltet Rollen und deren Bedeutung im System, welche Dokumente darf die Rolle xy sehen und oder schreiben, konfiguriert über die roleset.xml im configservice
system	Schema lesen, schreiben validieren
index	legt beim Ersten Start den Index an, schreibt Daten ins elasticsearch
search	service für alle Anfragen an elasticsearch, übersetzt die an cmis-sql angelegte Sprache in Anfragen an elasticsearch, reichert Anfragen mit Konditionen an
`async textextraction`	
controller	liest und schreibt Nachrichten von message queues, wertet die success und error queues aus in die der textextractor seine Ergebnisse schreibt
textextractor	versucht Text aus Dokumenten wie pdf, office, emails etc. zu extrahieren, wie contentanalyzer aber asynchron, ruft Dokument im System ab um es zu analysieren
`enaio client`	
rollator	übersetzt eine kleine Menge der redline rest-api Anfragen in die entsprechenden cloud API Funktionen

enaio service	Funktion
clientservice	wie in redline, service der den client ausliefert

enaio und k8s

Für jeden enaio service gibt es ein Docker image.

k8s services

Es werden Kuberentes Services benötigt für:

- configservice
- discovery
- authentication service

Alle enaio services brauchen den configservice. Deshalb muss der configservice unter einer festen Adresse im Cluster erreichbar sein. Dazu wird der Kubernetes service benutzt. Das Gleiche gilt für den discovery service. Die enaio services können untereinander kommunizieren, wenn sie die Adressen der jeweils anderen services von der discovery bekommen. Dazu müssen sich die enaio services zunächst an der discovery registrieren. Es muss eine feste Adresse im Cluster geben unter der die discovery erreichbar ist.

Der authentication service soll nach außen im Internet erreichbar sein. Dies wird mit einem ingress realisiert. Der Ingress leitet Anfragen von außen an einen kubernetes service im Cluster.

init container

Jeder enaio Pod bis auf den configservice hat einen Init Container, der das folgende kleine Skript ausführt, um zu prüfen ob der configservice verfügbar ist:

```
until wget -q --spider http://configservice/manage/health; do echo waiting for configservice; sleep 11; done;
```

Ein Init Container ist ein speziell ausgezeichneter Container, der vor den Containern in einem Pod ausgeführt wird. Erst wenn sich der Init Container erfolgreich beendet hat werden die weiteren Container im Pod gestartet.

liveness probe

Des weiteren wird in den deployments der enaio services eine liveness probe definiert.

```
    livenessProbe:
       httpGet:
          path: /manage/info
       initialDelaySeconds: 600
```

Im Standard wird ein Pod nach 3 fehlgeschlagenen Aufrufen von Kubernetes neu gestartet. Hier sollte nicht der Endpunkt "/manage/health" verwendet werden, da es zu einer Kettenreaktion kommen kann, wenn ein health check eines Services von weiteren Komponenten abhängt. Der health check des index service z.B. hängt

von elasticsearch ab, wenn elasticsearch nicht erreichbar ist, ist auch der health check des index service nicht erfolgreich. Kubernetes würde dann den index service neu starten, obwohl der service selbst kein problem hat.

Mit `initialDelaySeconds` kann ein delta in ms konfiguriert werden, bevor die livenessprobe nach dem Start des Pods ausgeführt wird. Der Wert ist in den Beispiel deployments recht hoch gewählt, da die services auf den configservice warten, dann das image herunterladen und dann erst der java Prozess startet. Ein zu kurz gewähltes Delta kann dazu führen, dass kein service ordentlich startet, da das System (CPU) unter Last ist (mehrere parallel startende jvms), was dazu führt, dass die Startzeit der einzelnen jvms steigt und die services nicht ordentlich hochfahren und unter "manage/info" nicht erreichbar sind wenn die livenessprobe Aufrufe starten. Als Ergebnis wird der pod neugestartet und die Schleife beginnt von vorn.

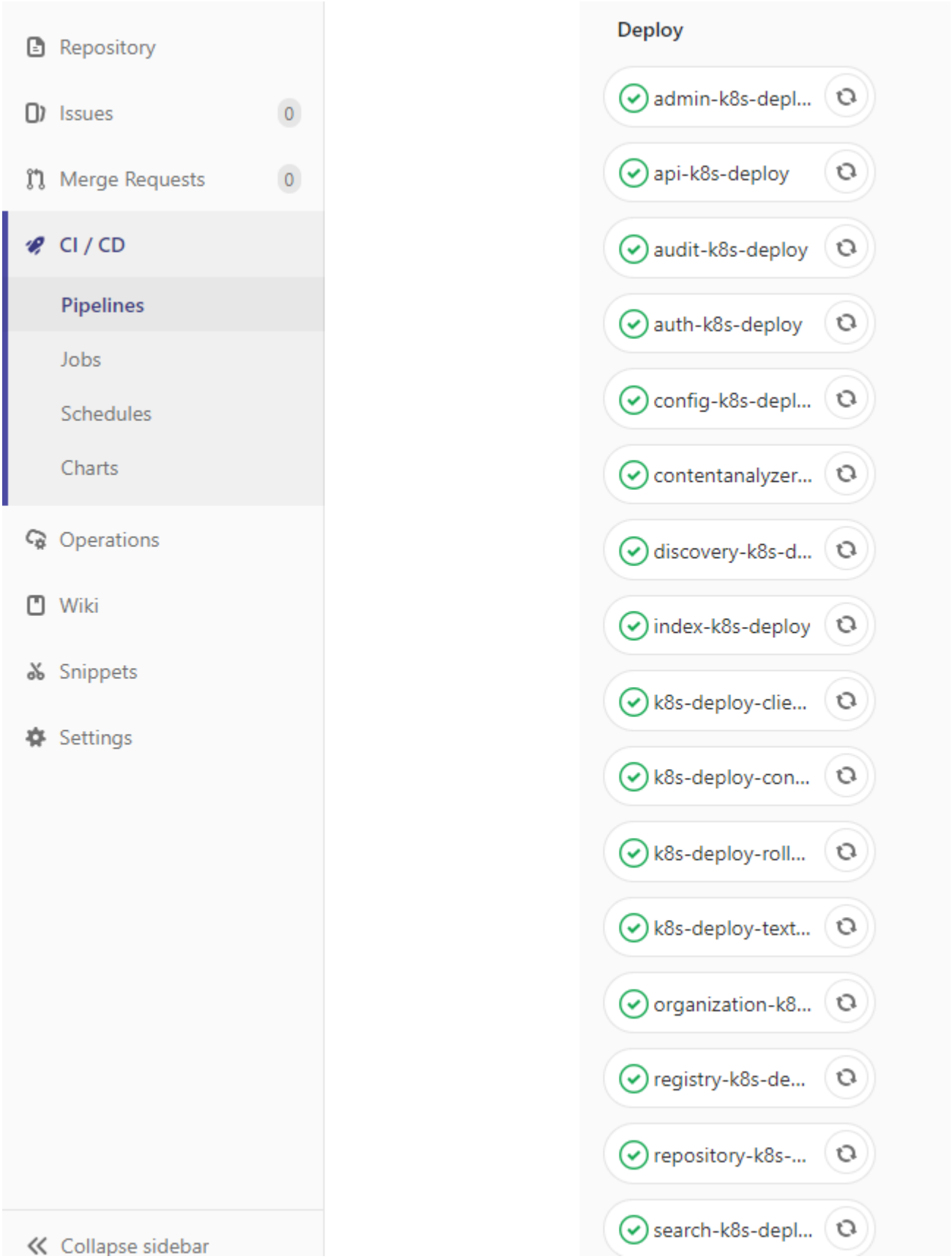
Des weiterin gibt es in Kubernetes noch readiness probes. Wenn eine readiness probe fehlschlägt werden keine Anfragen mehr an den Pod geleitet, bis die readiness probe wieder erfolgreich ist.

gitlabci

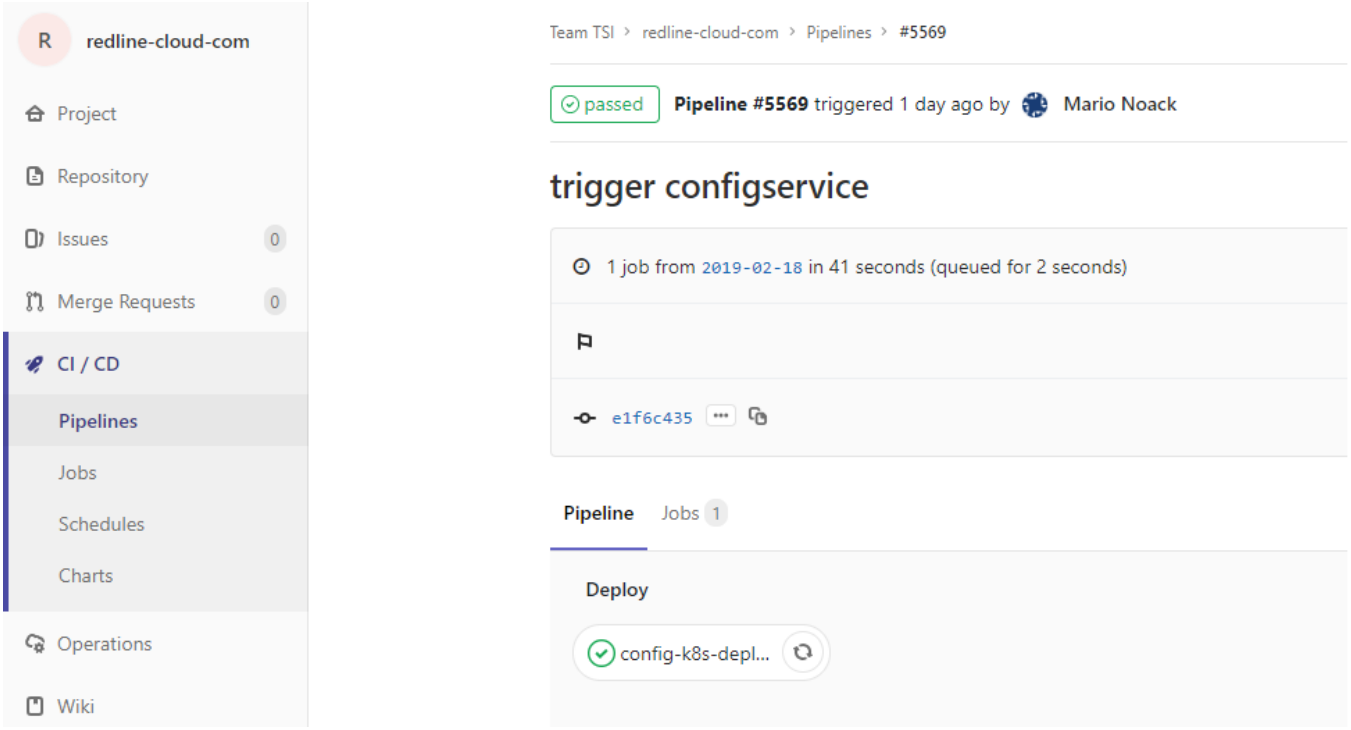
Die enaio cloud Komponenten werden in einen eigenen Kubernetes namespace installiert. Der Namespace entspricht dem Branch im `redline-cloud-com` git. Außerdem wird der Branch Name als docker Tag verwendet um die images aus der docker registry `docker.optimal-systems.org` herunter zu laden. Es wird ein Ingress erstellt, der dafür sorgt, dass die Installation unter https://"branchname".redline-cloud.com erreichbar ist.

Als Vorlage wurde die .gitlab-ci.yml für das enaioci System verwendet und angepasst.

In der gitlab-ci.yml gibt es für jeden enaio service, der im cluster deployed werden soll einen Job. Die Jobs sind so konfiguriert, dass sie nur gestartet werden wenn sich Dateien unter dem Pfad des jeweiligen services ändern z.B. `enaio/k8s/configservice`. Bei einem Initialen erstellen des Branches am gitlab Server werden alle Jobs ausgeführt, da für die gitlabci alle Dateien neu(geändert) sind.

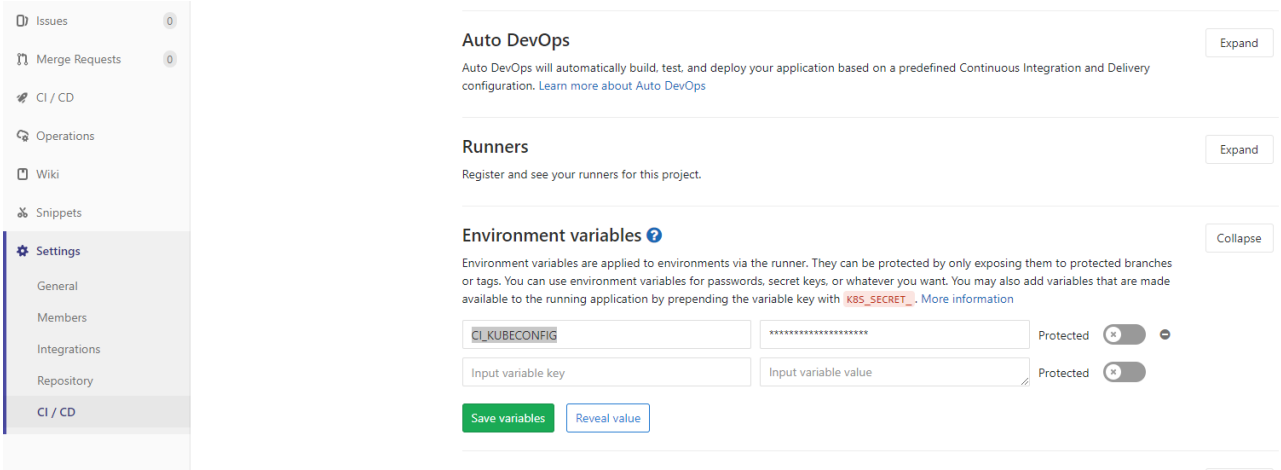


Danach werden nur die enaio services deployed, bei denen die Dateien angepasst wurden.



Die einzelnen Jobs rufen als erstes `kubectl apply -f namespace.yml` um den namespace zu erstellen, in den deployed werden soll. Dabei kann es zu Problemen bei apply kommen. Der Befehl apply erstellt ein Kubernetes Objekt, falls es noch nicht vorhanden ist, oder updated es, falls es bereits vorhanden ist. Dazu prüft der kubectl Client ob das Objekt vorhanden ist und führt dann den entsprechenden Aufruf am Kubernetes Api Server aus. Bei mehreren parallelen Aufrufen kann dies zu Fehlern führen. Mehrere Clients versuchen den namespace mit create parallel zu erstellen. Daher wird in den gitlab-ci Job der Befehl möglicherweise 2x ausgeführt. `if ! cat enaio/k8s/namespace.yaml | envsubst | kubectl apply -f -; then cat enaio/k8s/namespace.yaml | envsubst '${NAMESPACE}' | kubectl apply -f -; fi`

Die kubectl client config Datei wurde dem gitlab git Projekt als Umgebungsvariable `CI_KUBECONFIG` base64 encodiert `cat config | base64 -w0 > base64conf` hinzugefügt.



Zur Ausführung der in der .gitlab-ci.yml konfigurierten Schritte wird das docker image `docker.optimal-systems.org/oktopus/hel/enaioci-k8s-dpl-client:latest` verwendet. Das zugehörige Dockerfile ist im redlince-cloud git unter enaio/docker/enaioci-k8s-dpl-client eingecheckt.

Es wurde ein Branch erstellt und damit das deployment gestartet:

```
git checkout -b 2019-02-18
```

```
git push --set-upstream origin 2019-02-18
```

Nach dem Erstellen des git Branches sollte automatisch im Cluster ein neuer namespace mit dem Namen des Branches `2019-02-18` deployed werden.

Namespaces im Cluster anfragen: `kubectl get ns`

Alle Pods im namespace `2019-02-18` anzeigen: `kubectl get po -n 2019-02-18`

enaio cloud init

Die enaio services werden wie in blue und redline über yml Dateien konfiguriert. Abhänging von den Profilen mit denen ein Service gestartet wird fragt der service beim Start Konfigurations Dateien am configservice an. (Beispiel Profile: es -> service fragt application-es.yml an) In den Konfigurations Dateien werden Eigenschaften wie die Datenbank, der Datenbank Nutzer, die Adresse von Elasticsearch, die Adresse von redis, die Queue welche der textextractor und der controllerservice verwenden etc. konfiguriert.

Damit das enaio System funktioniert müssen die config Dateien entsprechend angepasst werden. Dazu wird ein Init containter im deployment des configservice verewendet, der das Skript `init.bash` (im redline-cloud git unter enaio/scripts) ausführt. Das Image von dem der init container gestartet wird ist `docker.optimal-systems.org/team-services/service-infrastructure/enaioci-config-init:latest` (im redline-cloud git unter enaio/docker/init-config)

Die Zugangsdaten für gitea, minio, keycloak und cockroach werden per secret Referenz an den container übergeben. Hier werden die secrets verwendet, die zuvor im namespace `infrastructure` erstellt wurden.

Das init script:

- erstellt einen client in dem bereits existierendem realm `hackathon` in keylcloak und konfiguriert die redirect Adressen
- ersetzt die Platzhalter in den config yml Dateien, die von den services verwendet werden mit den passende Werten

Config Datei	**zu ersetzende Werte**
application-oauth2.yml	clientsecret, admin user, admin password
application-dbscockroach.yml	cockroach user
application-storage.yml	minio access key, minio access secret
application-lc.yml	name für die queues die von dieser Instanz verwendet werden soll
systemHookConfiguration.json	name für die queue die das api gateway in dieser Instanz verwendet werden soll

- erstellt einen branch im gitea git und commited die config Dateien, sowie das Schema und die roleset.xml

Die yml Dateien mit Platzhaltern, das Schema, systemHookConfiguration.json und die roleset.xml (im redline-cloud git unter enaio/enaio-config) werden als configmaps von dem gitlabci Job in den namepsace deployed und per volume mount in den init container eingebunden.

Problem während des Termins

Das init script im configservice ist während des Termins fehlgeschlagen.

Um das Problem zu analysiern wurde ein deployment erstellt, welches dasselbe Image wie der init-container verwendet und die selben volume mounts hat. `kubectl apply -f dplIC.yaml`

In der Analyse hat sich gezeigt, dass das Passwort für keycloak und git in den http Aufrufen im init Skript nicht richtig encodiert wurde. Daher wurden die Passwörter angepasst.

Im init Skript wird curl für die Aufrufe an keycloak verwendet. Um das Problem zu lösen können bei curl die Werte mit `curl --data-urlencode $Wert` passend enkodiert werden.

Für den http Aufruf an git gibt es derzeit noch keine Alternative.

enaio cloud delete

Wenn auf dem git branch im redline-cloud git ein commit mit der commit message `delete namespace` erstellt wird, werden die enaio cloud Komponenten gelöscht.

Dazu wird ein Kubernetes job erstellt der das Skript `deleteyourself.bash` (im redline-cloud git unter enaio/scripts) ausführt. In diesem Skript wird:

- der git branch mit den config Dateien im gitea gelöscht
- der client aus dem keycloak gelöscht
- ein http delete am kubernetes api server ausgeführt um den namespace zu löschen

Wird in Kubernetes ein namespace gelöscht, werden auch alle Kubernetes Objekte in diesem namespace gelöscht.

Damit der http Aufruf am kubernetes Api server ausgeführt werden kann wird ein `role` Objekt und ein `role-binding` Objekt erstellt. Die Rolle eralaubt den Zugriff auf die Resource namespace mit dem Namen des git branches (hier `2019-02-28`). Das `role-binding` Objekt weist dem default service account des namespaces, welchen jeder Pod im namespace als Standard bekommt, die neue Rolle zu.