Frontend service
node -- angular, react oder ähnliches
Zeigt die Userdaten an
Greift über das API Service auf die DB zu
Authentifiziert sich am Keycloak Service
User muss sich am Frontend anmelden

API Service
API Service der einzige Service der direkt auf die DB und auf Keycloak zugreifen kann
Frontend und Producer authentifizieren sich über die API am Keycloak

DB Service
Wir wollen für die Persistenz Postgres in Exoscale verwenden
Producer schreibt über API Service die Daten nach Postgres und Frontend Service holt die Daten über API ab

Keycloak
Eine Keycloak Instanz für alle Kunden läuft in K8s
Wird über ArgoCD deployed

CI/CD Pipeline
Es soll Github Actions und die Github Container Registry verwendet werden.
K8s pullt beim Eintreffen eines neuen Images in der Registry automatisch die neue Version und startet neue Pods.

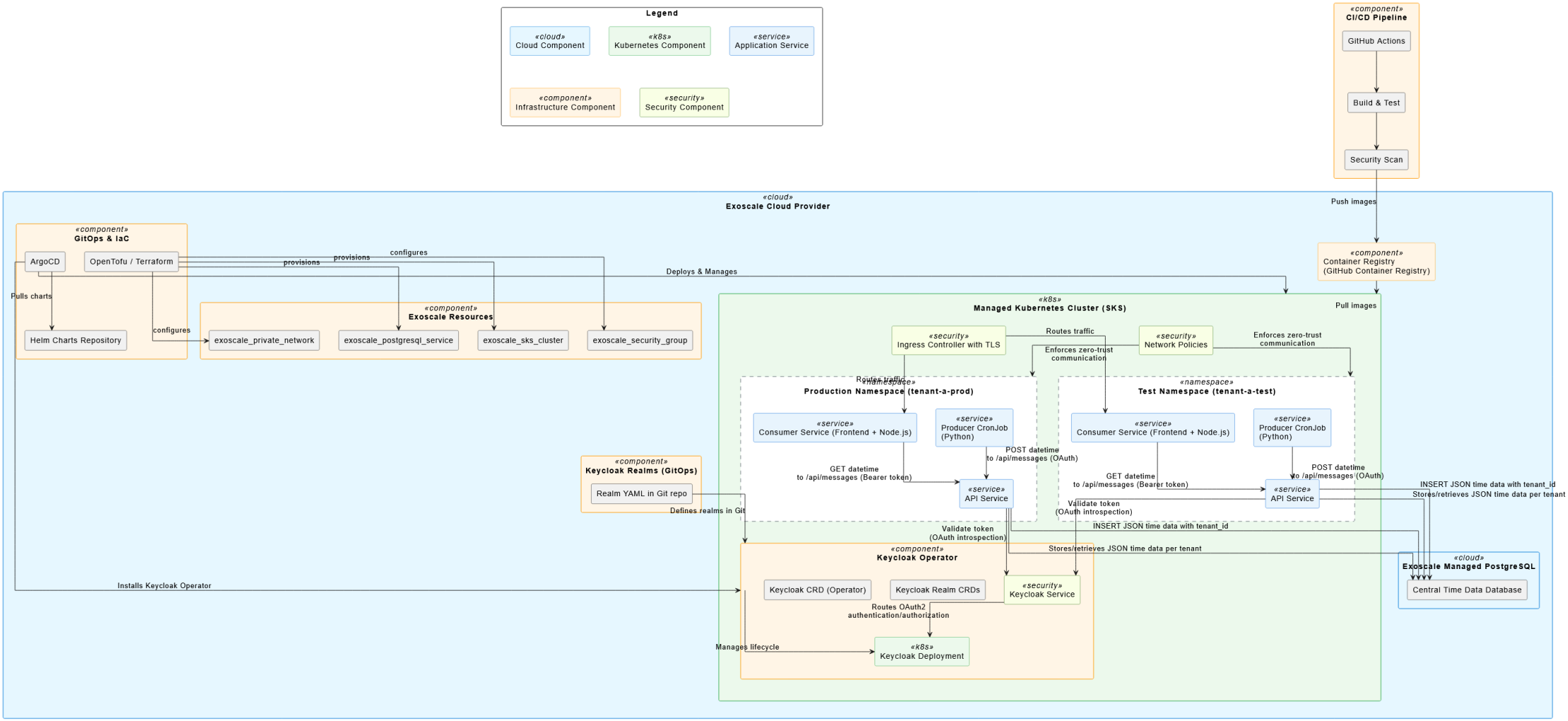
IaC
Es soll Opentofu verwendet werden. Dieses deployed das K8S Cluster, ArgoCD im K8s Cluster, sowie Postgres in Exoscale als PaaS.

ArgoCD

ArgoCD deployed Keycloak in einem eigenen Namespace in K8s.
Anschließend werden pro Kunde 2 Namespaces in K8s erstellt:
Prod und Test

In die Namespaces werden je deployed:
Frontend Service, API Service, Producer Service in K8s.
Zusätzlich wird ein Ingress Controller deployed.

INEP PT - Schütz Konzept Gruppe 01: Beier, Peer, Prugger



Automatisierter GitOps-Prozess zur Bereitstellung von Keycloak-Tenant-Realms

---

1. Git Commit des Tenant Realm YAML

Was passiert:

- Ein Entwickler erstellt oder ändert eine Datei des Typs `KeycloakRealm` im Verzeichnis `helm-charts/keycloak-realms/` eines Git-Repositories.
- Diese Datei definiert die Konfiguration eines neuen oder bestehenden Mandanten (Realm), inklusive Benutzer, Clients, Rollen und Berechtigungen.
- Sobald der Commit durchgeführt und gepusht wird, löst dies ein **Git Push Event** aus – dargestellt im Diagramm als `Git Push: Tenant Realm YAML`.

Wie das funktioniert:

- Der Git Push erzeugt ein **CloudEvent** (z. B. via webhook oder durch ein Polling von ArgoCD), welches als Triggersignal dient.
- Diese Konfiguration ist deklarativ, also "source of truth" – was im Git liegt, wird im Cluster reproduziert.

---

2. ArgoCD Sync (GitOps Mechanismus)

Was passiert:

- **ArgoCD** ist so konfiguriert, dass es kontinuierlich das Git-Repository überwacht.
- Sobald es einen neuen Commit erkennt, synchronisiert es automatisch den Zustand im Git mit dem Kubernetes-Cluster.

Wie das funktioniert:

- ArgoCD zieht die neue Konfiguration (Realm YAML) aus dem Git-Repository.
- Es führt die entsprechenden `kubectl apply`-Befehle aus und sorgt dafür, dass die Ressourcen im Cluster mit dem Git-Zustand übereinstimmen.
- Im Fall der Keycloak-Realm-Definition bedeutet dies: das Kubernetes Custom Resource (CR) `KeycloakRealm` wird im Cluster erstellt oder aktualisiert.

---

3. Operator-Reconciliation (Keycloak Operator greift ein)

Was passiert:

- Der **Keycloak Operator**, der im Cluster als Kubernetes Controller läuft, erkennt das neue oder geänderte KeycloakRealm-Objekt.

Wie das funktioniert:

- Der Operator beobachtet Custom Resources (CRDs) wie KeycloakRealm.
- Sobald eine Änderung erkannt wird, verwendet der Operator die **Keycloak Admin API**, um den tatsächlichen Zustand im Keycloak-System an den deklarierten Zustand im Git-Repo anzupassen.
- Dabei werden:
  - Neue Realms angelegt
  - Clients (z. B. api-service, consumer) registriert
  - Rollen, Benutzer und Berechtigungen angelegt oder angepasst

Technisch:

- Es erfolgt eine Reconciliation-Schleife: Desired State (im YAML definiert) ≙ Actual State (im Keycloak-System)
- Dieses Prinzip ist typisch für Kubernetes-Controller.

---

4. Anwendungsfunktionalität aktiv (Application Continuity)

Was passiert:

- Die neuen oder aktualisierten Keycloak-Realms stehen sofort zur Verfügung.
- Dienste wie Producer, API Service und Consumer können nun Zugriffstokens (OAuth2) für den neuen Realm erhalten und damit kommunizieren.

Wie das funktioniert:

- Die **Produzenten** senden mit OAuth-Tokens Daten an /api/messages.
- Der **API Service** validiert das Token über den Keycloak-Service (introspection).
- Die Daten (z. B. Zeitstempel-Informationen) werden pro Tenant in der zentralen PostgreSQL-Datenbank gespeichert.
- **Konsumenten** können mit Bearer-Tokens diese Informationen wieder abrufen.

Ergebnis:

- Vollständig multitenant-fähiger Betrieb, sofort nach YAML-Deployment, ohne manuelle Eingriffe.

---

5. CI/CD und Infrastrukturbereitstellung (paralleler Prozess)

Was passiert:

- Während die Realm-Provisionierung GitOps-gesteuert erfolgt, läuft parallel eine klassische CI/CD-Pipeline für Servicecode-Änderungen.

Wie das funktioniert:

- Änderungen am Code eines Services (z. B. Producer oder API) starten via **GitHub Actions**:
  - Unit Tests und Builds (Build & Test)
  - Sicherheitsprüfungen (Security Scan)
  - Deployment der Container-Images in die Registry
- Die Images werden anschließend im Cluster durch **ArgoCD** deployed, da auch diese über Helm-Charts verwaltet werden.

🔄 Zusammenfassung der Prozesslogik

Schritt	Komponente	Aufgabe
1	Entwickler + Git	Änderung des KeycloakRealm-YAML, Git Push
2	ArgoCD	Sync vom Git-Repo → Kubernetes CR wird erstellt
3	Keycloak Operator	Erkennung des neuen CR, Konfiguration des Realms via Admin API
4	Dienste (API etc.)	Nutzung der neuen Realm-Konfiguration für Authentifizierung
5	CI/CD	Parallel laufende Image Builds, Tests, Pushes und Deployments

🛠️ Technisches Cheatsheet – K8s Keycloak Architecture

🔑 Keycloak

Komponente	Beschreibung
Keycloak Operator	Kubernetes-native Verwaltung von Keycloak-Instanzen via CRDs
KeycloakRealm CRD	Definiert Realm, Clients, Rollen etc. als YAML
Keycloak Deployment	StatefulSet oder Deployment innerhalb des Clusters
Keycloak Service	Interner OAuth2-Endpunkt zur Tokenvalidierung (introspection)

Kubernetes-Setup

Komponente	Beschreibung
Namespaces (tenant-a-prod/test)	Mandantenisolation für Dev/Prod durch Namespace-Segmentierung
Consumer / Producer Services	Frontend (Node.js) + Producer (Python CronJob)
API Service	OAuth2-geschützte REST-API für Zeitdaten

Sicherheit & Netzwerke

Komponente	Beschreibung
Ingress Controller + TLS	TLS-Termination & Routing per Hostname/Pfad
Network Policies	Zero-Trust-Kommunikation (whitelist-based) zwischen Pods
OAuth2 Token Handling	Producer/Consumer → API mit Bearer/OAuth2 Tokens

GitOps-Prozess

Schritt	Beschreibung
1. Git Push	Realm-YAML wird ins Git-Repo gepusht
2. ArgoCD Sync	Erkennt Commit, synchronisiert Helm Chart
3. Operator Action	Beobachtet CRDs, provisioniert Realm in Keycloak
4. Tokenbereitstellung	Neue Clients können sofort OAuth2-Token holen
5. CI/CD	Build, Test, Security Scan, Image Push

Cloud-Komponenten (Exoscale)

Komponente	Beschreibung
Managed Kubernetes Cluster (SKS)	Container-Orchestrierung
Managed PostgreSQL	Multi-Tenant Zeitdatenbank
Exoscale Resources	IaC verwaltete Ressourcen: Cluster, Netzwerke, SGs
Private Network	Isoliertes Netzwerk für SKS und Datenbank

⚙️ Infrastructure as Code & CI/CD

Tool	Zweck
OpenTofu (Terraform Fork)	Provisioniert Exoscale-Infrastruktur
ArgoCD	GitOps-Controller für Deployment
Helm	Package-Manager für K8s-Ressourcen
GitHub Actions	CI/CD Pipeline: Build/Test/Security
Container Registry	Speicherung fertiger Images

🔍 Tokenvalidierung Flow

1. Client (Producer/Consumer) → API mit Bearer-Token
2. API → Keycloak Service (OAuth2 Introspection)
3. Erfolgreiche Validierung → Zugriff auf Datenbank (pro tenant_id)

🧱 Mandantenfähige Datenhaltung

Designmerkmal	Umsetzung
Mandantenisolation	K8s Namespace pro Tenant
Zeitdaten	JSON-Schema mit tenant_id
Zentrale DB	PostgreSQL (Mandanten durch tenant_id logisch getrennt)