



UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Elaborato per l'esame di Network Security

*Secure Backend Development on
Kubernetes: A DevSecOps
Approach with GitLab CI/CD*

Anno Accademico 2024/2025

Docente
Simon Pietro Romano

Autori
Francesco Scognamiglio M63001364
Felice Micillo M63001377

Abstract

Il presente lavoro descrive la progettazione e la realizzazione di un'infrastruttura backend sicura e modulare basata su *Kubernetes*, con un'attenzione particolare all'integrazione automatica dei controlli di sicurezza lungo l'intero ciclo di sviluppo. L'architettura segue il paradigma a microservizi e combina diversi componenti chiave: un backend sviluppato in *Node.js* con *Express*, il sistema di gestione delle identità *Keycloak*, un database *PostgreSQL* per la persistenza dei dati, *Vault* per la gestione sicura dei segreti, e una suite di monitoraggio basata su *Prometheus* e *Grafana*.

Il cuore del progetto è una pipeline *CI/CD* completamente automatizzata, realizzata su *GitLab*, pensata per incorporare verifiche di sicurezza fin dalle prime fasi dello sviluppo. Ogni commit avvia una serie di controlli: analisi statica del codice (*SAST*) con *ESLint* e *NodeJs-Scan*, scansione delle immagini Docker con *Trivy*, verifica dei manifest *Kubernetes*, build e pubblicazione delle immagini, deployment automatizzato tramite *Helm*, e test dinamici (*DAST*) con *OWASP ZAP*. Il processo è strutturato per fermarsi in caso di vulnerabilità gravi,

richiedendo interventi correttivi prima di procedere al rilascio.

Per garantire visibilità completa sullo stato di sicurezza del sistema, i risultati prodotti da ciascuna fase vengono raccolti e aggregati in un report centralizzato. Questo include vulnerabilità rilevate a livello di codice, di runtime e di infrastruttura, grazie anche all’impiego di *Kubescape* per l’analisi della postura di sicurezza del cluster *Kubernetes*. Ogni job produce artefatti utili alla tracciabilità e all’audit delle attività svolte.

Dal punto di vista infrastrutturale, l’architettura adotta misure di sicurezza avanzate: l’uso di *NetworkPolicy* per limitare le comunicazioni tra i pod, configurazioni di *SecurityContext* restrittive, gestione automatizzata dei certificati *TLS* con *Cert-Manager* e un controllo sicuro degli ingressi verso i servizi esposti, supportato da aggiornamenti *DNS* dinamici. Completa il sistema un modulo di rilevamento intrusioni basato su *Tetragon*, capace di monitorare eventi anomali a livello kernel – come l’apertura di shell o l’uso sospetto di socket – e inviare notifiche in tempo reale via *Telegram*. Gli eventi vengono inoltre registrati su *Loki*, offrendo strumenti di analisi post-incidente.

Nel complesso, il progetto propone una soluzione concreta, osservabile e completamente automatizzata, in cui i principi del *DevSecOps* trovano applicazione pratica attraverso strumenti e configurazioni capaci di migliorare sensibilmente la resilienza e la sicurezza dell’ambiente cloud-native.

Indice

Abstract	i
1 Introduzione	1
1.1 Obiettivo	2
1.2 Tecnologie e approccio DevSecOps	3
2 Architettura del Sistema	6
2.1 Panoramica del cluster Kubernetes	7
2.1.1 Kubernetes	7
2.1.2 Architettura	8
2.2 Helm Chart	10
2.3 Pod principali dell'applicazione	12
2.3.1 Express	13
2.3.2 Keycloak	18
2.3.3 Database	21
2.4 Comunicazione interna tra i servizi	22
2.5 Componenti di supporto	24
2.5.1 Nginx	25
2.5.2 Monitoring	26
2.5.3 Cert Manager	28

2.5.4	Vault e Vault CSI	29
3	Sicurezza dell'Infrastruttura	30
3.1	SecurityContext nei deployment	31
3.1.1	Express	32
3.1.2	Keycloak	33
3.1.3	Database	33
3.2	NetworkPolicy applicate ai pod	34
3.2.1	Default-Deny-All	35
3.2.2	Accesso da Prometheus e Grafana	35
3.2.3	Express	36
3.2.4	Keycloak	37
3.2.5	Database keycloak-db	38
3.2.6	Database app-db	39
3.3	Configurazione dell'Ingress Controller	40
3.3.1	Configurazione di NGINX	41
3.3.2	Ingress per il Servizio Express	42
3.3.3	Accesso sicuro ai pannelli admin	43
3.4	Gestione dei segreti con Vault e Vault-CSI	45
3.4.1	Accesso controllato tramite ServiceAccount	46
3.4.2	SecretProviderClass (SPC)	46
3.4.3	syncSecret	47
3.5	Certificati SSL tramite Cert-Manager	48
3.5.1	ClusterIssuer e Certificate	48
3.5.2	Integrazione con Vault per le credenziali Cloudflare	49

4	SecDevOps	51
4.1	Cos'è il SecDevOps?	52
4.2	Pipeline GitLab CI/CD	53
4.2.1	Analisi statica del codice (SAST)	54
4.2.2	Container Scanning	55
4.2.3	Build e Push	57
4.2.4	Validazione dei manifesti e linting	59
4.2.5	Deploy	61
4.2.6	Kubernetes Security	62
4.2.7	DAST – Dynamic Application Security Testing	64
4.3	Raccolta dei Report	66
4.4	Esecuzione e Mitigazione	68
4.4.1	Fase SAST – Errori rilevati da ESLint	69
4.4.2	Fase SAST – Errori rilevati da NodeJsScan	70
4.4.3	Container Scanning – Immagine Keycloak	72
4.4.4	Kubernetes Security – Scansione con Kubescape	72
4.4.5	Dynamic Application Security Testing (DAST)	74
5	Intrusion Detection e Sicurezza Runtime	76
5.1	Tetragon: Sicurezza Runtime con eBPF	77
5.1.1	Tecnologia eBPF	77
5.1.2	Architettura di Tetragon	78
5.1.3	Politiche di Tracciamento (TracingPolicy)	81
5.2	Integrazione con Cilium: Sicurezza e Visibilità di Rete	82
5.3	Monitoraggio degli Accessi Runtime	83
5.3.1	Obiettivo del Monitoraggio	84

5.3.2	Implementazione della Soluzione	85
5.3.3	Risultati Ottenuti	85

Capitolo 1

Introduzione

Negli ultimi anni, l'adozione delle architetture a microservizi e l'uso dei container gestiti con Kubernetes hanno cambiato profondamente il modo in cui le applicazioni vengono progettate, distribuite e mantenute. Questa trasformazione ha portato vantaggi evidenti, come una maggiore scalabilità, resilienza e flessibilità nei cicli di sviluppo. Tuttavia, ha anche aperto la porta a nuove sfide sul fronte della sicurezza. In ambienti distribuiti e dinamici come quelli basati su Kubernetes, la superficie d'attacco si espande notevolmente. Aspetti come la comunicazione tra microservizi, la gestione dei segreti, l'accesso ai dati sensibili o la configurazione del cluster richiedono un'attenzione continua. Inoltre, se l'automazione dei processi di build e deploy non è accompagnata da controlli di sicurezza efficaci, può diventare un punto debole per l'intera infrastruttura.

1.1 Obiettivo

L'obiettivo principale di questo progetto è la realizzazione di un'infrastruttura backend sicura, basata su Kubernetes, che integri fin dalle prime fasi dello sviluppo strumenti e pratiche orientate alla sicurezza, in linea con i principi del modello *DevSecOps*.

A partire da un'applicazione web containerizzata, il lavoro è articolato sui seguenti step:

- **Progettazione di un'architettura a microservizi**, isolata e logicamente segmentata tramite l'utilizzo di namespace *Kubernetes*, per garantire una maggiore separazione e controllo tra i diversi componenti del sistema.
- **Implementazione di un sistema di autenticazione centralizzato utilizzando Keycloak**, con l'obiettivo di gestire identità e accessi in maniera sicura e coerente.
- **Configurazione sicura del cluster Kubernetes**, applicando le principali *best practice* di sicurezza: definizione di policy di rete restrittive, adozione del principio del minimo privilegio nei permessi, segregazione dei segreti, utilizzo del protocollo *TLS* con certificati digitali, e impostazione di parametri di sicurezza nei manifesti di deployment.
- **Integrazione di strumenti avanzati** come *Vault*, *Prometheus*, *Grafana*, *Cert-Manager* per la gestione centralizzata dei segreti,

il monitoraggio delle metriche e l'automazione nella generazione e gestione dei certificati.

- **Analisi automatizzata della sicurezza del cluster**, sfruttando tool in grado di rilevare vulnerabilità sia nella configurazione dell'ambiente che nei container impiegati.
- **Implementazione di una pipeline CI/CD basata su GitLab**, che include fasi di scansione automatica del codice sorgente (*SAST*) e delle API esposte (*DAST*), con l'obiettivo di assicurare un livello di sicurezza costante nel tempo, lungo tutto il ciclo di vita dell'applicazione.
- **Sistema di intrusion detection basato su Tetragon**, capace di monitorare in tempo reale attività sospette a livello kernel, come l'apertura di shell non autorizzate o la creazione di socket anomali, integrandosi con strumenti di logging e notifica per una risposta immediata.

1.2 Tecnologie e approccio DevSecOps

L'architettura del progetto si fonda su Kubernetes, la piattaforma di orchestrazione di container che consente di gestire in modo scalabile il deployment, la configurazione e il monitoraggio di applicazioni distribuite. La configurazione del cluster è stata gestita principalmente con

Helm, strumento che ha facilitato l'utilizzo e la personalizzazione di chart ufficiali per l'installazione di componenti fondamentali:

- **NGINX Ingress Controller**, utilizzato per esporre il backend all'esterno in modo sicuro;
- **Express in NodeJS**, per la realizzazione di un web server leggero e performante;
- **Keycloak**, per la gestione centralizzata di autenticazione e identità degli utenti;
- **Vault e Vault-CSI**, per una gestione sicura e dinamica dei segreti e delle credenziali sensibili;
- **Cert-Manager**, impiegato per l'emissione e il rinnovo automatico di certificati SSL;
- **Prometheus e Grafana**, strumenti adottati per il monitoraggio e la visualizzazione delle metriche del sistema;
- **PostgreSQL**, usato come database di persistenza sia per l'applicazione backend sia per Keycloak.

Per integrare la sicurezza in ogni fase del ciclo di vita del software, è stata progettata una pipeline CI/CD su GitLab, che prevede controlli automatici sia sul codice sorgente (SAST) sia sulle API esposte (DAST). Questo approccio consente di individuare tempestivamente

eventuali vulnerabilità, rendendo i controlli ripetibili e meno soggetti a errori manuali.

L'infrastruttura è stata configurata seguendo i principi del security by design, adottando misure come:

- **la definizione di securityContext nei manifest** dei pod per rafforzare le politiche di esecuzione sicura;
- **l'applicazione di NetworkPolicy**, per limitare la comunicazione tra pod secondo criteri predefiniti;
- **la gestione dei segreti tramite Vault**, in modo da evitare la loro esposizione diretta nei file di configurazione;
- **la separazione logica** dei componenti dell'applicazione attraverso namespace distinti;
- **la configurazione di Ingress e certificati SSL**, per proteggere le comunicazioni esterne con un layer crittografico affidabile.

Capitolo 2

Architettura del Sistema

In questo capitolo viene esaminata l'architettura dell'infrastruttura realizzata, con un focus specifico sulla suddivisione in namespace, sull'organizzazione dei pod principali e sulle modalità di comunicazione tra i diversi componenti. L'intero sistema è stato progettato per operare su un cluster *Kubernetes*, sfruttando appieno le funzionalità offerte dalla piattaforma in termini di gestione, scalabilità e sicurezza delle applicazioni containerizzate.

Il cuore dell'architettura è rappresentato da un'applicazione backend sviluppata con *Node.js* ed *Express*, che funge da punto di accesso principale al sistema ed interagisce con un servizio di autenticazione centralizzato basato su *Keycloak* e con database dedicati.

A supporto dell'infrastruttura sono stati integrati diversi componenti, distribuiti in namespace separati per garantire una maggiore modularità e sicurezza.

2.1 Panoramica del cluster Kubernetes

2.1.1 Kubernetes

Kubernetes è una piattaforma open-source per l'orchestrazione di container, progettata per automatizzare il deployment, la gestione, il bilanciamento del carico e il monitoraggio di applicazioni containerizzate. Originariamente sviluppato come progetto interno da Google, è oggi considerato lo standard de facto nella gestione di infrastrutture cloud-native moderne.

Uno degli elementi chiave di Kubernetes è il suo approccio dichiarativo: gli utenti definiscono lo stato desiderato dell'infrastruttura tramite file di configurazione in formato *YAML*, mentre il sistema si occupa di raggiungerlo e mantenerlo in maniera automatica. Questa modalità di gestione, insieme alla capacità di operare in ambienti dinamici e distribuiti, rende Kubernetes particolarmente adatto allo sviluppo di applicazioni scalabili e resilienti.

La piattaforma consente di astrarre l'infrastruttura fisica sottostante attraverso oggetti come *Pod*, *Service*, *Deployment*, *Ingress* e *Namespace*, fornendo un elevato livello di controllo e isolamento tra i componenti. Inoltre, mette a disposizione strumenti integrati per la gestione della sicurezza, della rete, dei segreti e per il monitoraggio dello stato delle applicazioni, facilitando così l'adozione di pratiche *DevOps* e *DevSecOps* in ambienti produttivi complessi.

2.1.2 Architettura

Di seguito è riportata una versione semplificata del cluster Kubernetes, suddiviso per namespace. Sono stati inclusi esclusivamente i pod principali, configurati e aggiunti manualmente all'infrastruttura.

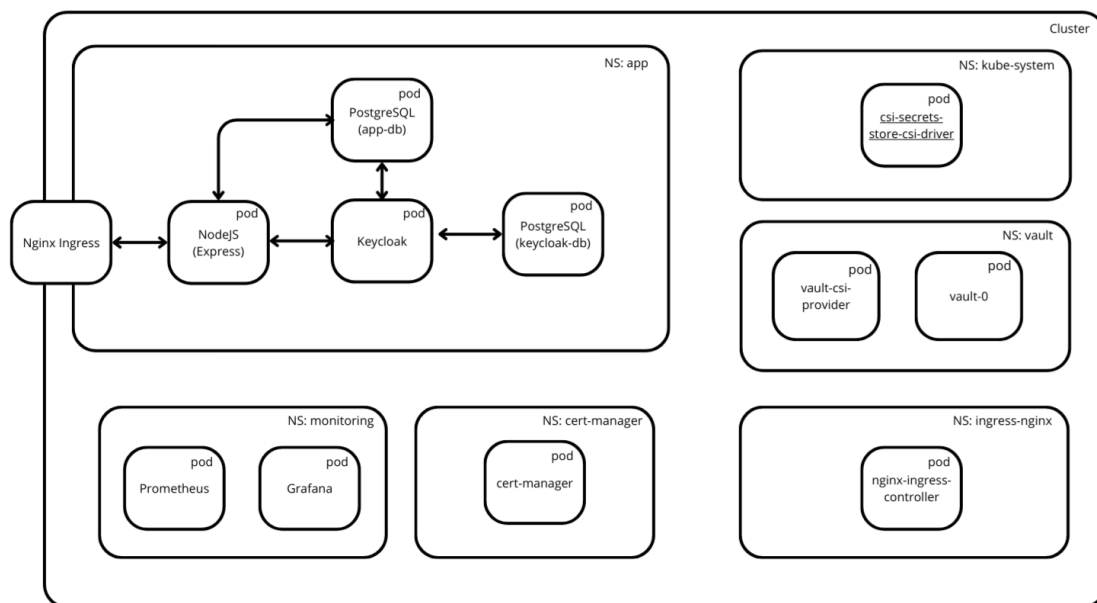


Figura 2.1: Diagramma Visivo del Cluster

L'immagine fornisce una rappresentazione semplificata del cluster Kubernetes, organizzata per namespace e focalizzata sui principali pod configurati manualmente. Al centro dell'architettura si trova il namespace **app**, che ospita i servizi applicativi principali. Il punto di ingresso al cluster è gestito dal controller *NGINX Ingress*, incaricato di ricevere le richieste esterne e indirizzarle verso un'applicazione Node.js sviluppata con Express, che costituisce il backend dell'infrastruttura.

Il pod Express comunica direttamente con un database *PostgreSQL*

chiamato *app-db*, utilizzato per la persistenza dei dati applicativi. Inoltre, interagisce con Keycloak, il servizio per la gestione centralizzata delle identità, che si appoggia su due database distinti: *app-db*, condiviso per eventuali integrazioni, e *keycloak-db*, riservato ai dati interni di autenticazione.

Accanto al namespace applicativo, sono stati definiti altri namespace specializzati per supportare le funzionalità trasversali del sistema:

- Il namespace monitoring ospita *Prometheus* e *Grafana*, strumenti fondamentali per la raccolta e la visualizzazione delle metriche di sistema.
- Nel namespace *cert-manager* è collocato l'omonimo componente, responsabile della generazione e del rinnovo automatico dei certificati TLS.
- La gestione dei segreti e della sicurezza è affidata a *Vault*, posizionato nel namespace vault, insieme al provider *Vault CSI*, che consente il montaggio sicuro dei segreti nei pod.
- Il supporto a questa integrazione è garantito dal CSI driver *secrets-store-csi-driver*, installato nel namespace kube-system.
- Infine, il namespace ingress-nginx ospita il controller NGINX Ingress vero e proprio, che applica le regole di routing del traffico in ingresso al cluster.

2.2 Helm Chart

Helm è uno strumento chiave nell’ecosistema Kubernetes, spesso definito come il package manager per Kubernetes. La sua funzione principale è semplificare e automatizzare il deployment, l’aggiornamento e la gestione di applicazioni complesse all’interno di un cluster.

Alla base del suo funzionamento c’è il concetto di chart, un pacchetto che include tutti i file di configurazione necessari per distribuire un’applicazione o un insieme di risorse Kubernetes. Ogni chart contiene template YAML parametrizzati, un file *values.yaml* per la definizione dei valori personalizzabili e un file *Chart.yaml* con le informazioni di base, come nome, versione e dipendenze del pacchetto.

Helm adotta un approccio dichiarativo e riutilizzabile alla definizione delle risorse, permettendo di installare un’intera applicazione – comprensiva di *Deployment*, *Service*, *Ingress*, *ConfigMap*, *Secret* e altri oggetti – con un singolo comando, semplicemente richiamando un chart e passando eventuali valori personalizzati.

Un aspetto particolarmente utile è la gestione dello stato delle release: Helm tiene traccia di ogni installazione, consentendo aggiornamenti incrementali, rollback in caso di problemi e un’analisi dettagliata delle modifiche applicate. Questo semplifica notevolmente la manutenzione e il ciclo di vita delle applicazioni distribuite.

Infine, uno dei maggiori punti di forza di Helm è la possibilità di gestire ambienti diversi (sviluppo, test, produzione) utilizzando la stessa

base di configurazione riducendo significativamente il rischio di errori manuali e supporta un approccio DevOps evoluto, ideale per progetti complessi.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Values.express.name }}
spec:
  replicas: {{ .Values.express.replicas }}
  selector:
    matchLabels:
      app: {{ .Values.express.app }}
  template:
    metadata:
      labels:
        app: {{ .Values.express.app }}
    spec:
      serviceAccountName: {{ .Values.express.serviceAccount }}
      initContainers:
        - name: copy-vault-secrets
          image: busybox:1.36
          command:
            - /bin/sh
            - -c
            - |
              cp /mnt/vault/* /mnt/runtime/
          volumeMounts:
            - name: vault-secrets
              mountPath: /mnt/vault
              readOnly: true
            - name: runtime-secrets
              mountPath: /mnt/runtime
      containers:
        - name: {{ .Values.express.name }}
          image: {{ .Values.express.image }}:{{ .Values.express.version }}
          ports:
            - containerPort: {{ .Values.express.targetPort }}
```

Figura 2.2: Esempio di un template per Helm

2.3 Pod principali dell'applicazione

L'architettura dell'applicazione si articola attorno a quattro pod principali, ciascuno progettato per svolgere un ruolo specifico e sinergico all'interno del sistema complessivo. Questi elementi costituiscono la spina dorsale dell'ambiente backend, contribuendo in maniera coordinata alla gestione dell'autenticazione, all'elaborazione delle logiche applicative e alla persistenza strutturata dei dati.

Il primo pod ospita l'applicazione *Express*, un backend sviluppato in *Node.js* che funge da punto di ingresso per tutte le richieste provenienti dall'esterno. Questo componente si occupa dell'elaborazione delle logiche principali, orchestrando le interazioni con i servizi interni, tra cui il sistema di autenticazione e il database, e garantendo l'esposizione sicura delle API, sia REST che GraphQL.

A fianco di Express, il secondo pod è dedicato a *Keycloak*, la piattaforma open-source per la gestione centralizzata delle identità e degli accessi. Si occupa di autenticare gli utenti e autorizzarne le azioni, controllando l'accesso alle risorse dell'intero ecosistema. Grazie a un'integrazione approfondita con Express, Keycloak consente una gestione granulare di utenti, ruoli e permessi, sia in modalità standard sia in contesti più avanzati come l'autenticazione passwordless.

Il terzo pod è dedicato al database di Keycloak, utilizzato per memorizzare tutte le configurazioni interne della piattaforma, inclusi i dettagli dei realm, gli utenti registrati, le politiche di accesso e le in-

formazioni necessarie al corretto funzionamento del sistema di identity management.

Infine, il quarto pod ospita il database dell'applicazione, che raccoglie e gestisce tutte le informazioni legate al dominio applicativo vero e proprio. Qui risiedono i dati relativi a profili utente, esperienze, configurazioni dinamiche e ogni altro elemento funzionale che fa parte della logica di business.

I paragrafi successivi approfondiranno ciascuno di questi pod, analizzandone nel dettaglio le funzionalità principali, le configurazioni chiave e le modalità di interazione con gli altri componenti del cluster Kubernetes, offrendo così una visione completa e strutturata dell'intero ecosistema applicativo.

2.3.1 Express

Il cuore logico dell'infrastruttura backend è costituito dall'applicazione *Express*, sviluppata in *Node.js*, che funge da punto di accesso alle richieste in arrivo nel cluster Kubernetes attraverso il controller Ingress. Non si tratta di un semplice server HTTP, ma di un nodo applicativo centrale, progettato per integrare autenticazione, monitoraggio, gestione API (sia REST che GraphQL) e una struttura sicura per il trattamento dei segreti.

L'avvio dell'applicazione è gestito tramite il file *server.js*, che inizializza le variabili d'ambiente con *dotenv* e carica in modo sicuro i segreti

montati nel pod grazie all'integrazione con Vault-CSI. Subito dopo, il server si avvia sfruttando una serie di middleware fondamentali: *cookie-parser* per i cookie, *cors* per abilitare le richieste cross-origin verso il frontend, ed *express.json()* per il parsing delle richieste in formato JSON. L'opzione *trust proxy* viene attivata per assicurare una corretta gestione delle richieste inoltrate da NGINX.

Uno degli elementi più versatili della piattaforma è l'integrazione con *GraphQL-Yoga*, che espone un endpoint moderno e facilmente testabile tramite interfaccia interattiva (in fase di sviluppo). Ogni richiesta GraphQL viene arricchita con informazioni sull'utente, ricavate da un token JWT presente nei cookie. Questo processo è gestito da una funzione dedicata, *getUserFromToken*, incaricata di validare il token e restituire i dati utente da includere nel contesto della query o mutazione.

Per garantire stabilità operativa, l'applicazione integra due moduli di monitoraggio autonomi: *DBMonitor* e *AUTHMonitor*, entrambi basati su *EventEmitter*. Il primo verifica la disponibilità del database eseguendo query di test con Prisma ORM, mentre il secondo interroga l'endpoint *.well-known* di Keycloak per assicurarne la corretta operatività. In caso di problemi, i middleware collegati a questi moduli possono bloccare temporaneamente le richieste, restituendo un errore 503 e impedendo così che anomalie nei servizi a valle impattino sul comportamento dell'API.

A supporto di questa logica c'è anche *RecoveryManager*, un modulo che rileva il ripristino dei servizi e sincronizza lo stato tra Keycloak e il database, operazione utile per mantenere coerenza e integrità nei dati utente. Questa sincronizzazione può essere estesa anche a meccanismi di ripristino o riconciliazione più complessi.

Dal punto di vista funzionale, l'applicazione espone una serie di rotte REST organizzate in moduli:

- **/auth:** gestisce autenticazione, registrazione, logout, verifica OTP, reset password e altre operazioni legate a Keycloak.
- **/fedauth:** per l'integrazione con sistemi federati (come OAuth con Google).
- **/user e /webauthn:** dedicate alla gestione dell'utente e all'autenticazione passwordless.

Accanto alle rotte REST, il modulo *resolvers.js* definisce le query e mutazioni GraphQL, protette da un sistema di autorizzazione che richiede la presenza di un token JWT valido per ogni operazione. Prisma ORM si occupa dell'interazione con il database PostgreSQL, sfruttando uno schema ben definito (*prisma.schema*) che modella entità come User, Experience, Category e SavedExperience.

```
//Yoga GraphQL
const yoga = createYoga({
  schema,
  context: async ({ request }) => {
    const cookieHeader = request.headers.get('cookie') || '';
    const cookies = Object.fromEntries(cookieHeader.split(';').map(c => {
      const [key, value] = c.trim().split('=');
      return [key, decodeURIComponent(value)];
    }));
    const token = cookies['accessToken'];
    const user = await getUserFromToken(token);
    return { user };
  },
  graphql: true
});

//Express
const app = express();
const port = process.env.EXPRESS_PORT || 4000;
app.set('trust proxy', true);
app.use(express.json());
app.use(cookieParser());
app.use(cors({
  origin: process.env.FE_HOSTNAME,
  credentials: true,
}));

//DB Monitor Health Check
const dbMonitor = new DBMonitor();
dbMonitor.start();
app.use(dbMonitor.middleware());
app.get("/health-db", (req, res) => {
  if (dbMonitor.isHealthy()) {
    res.status(200).json({ status: "ok", db: "online" });
  } else {
    res.status(503).json({ status: "degraded", db: "offline" });
  }
});
```

Figura 2.3: Estratto di codice dal file server.js

Dockerfile

L'immagine Docker dell'applicazione è costruita a partire dalla base ufficiale *node:lts-alpine3.20*, una scelta mirata per ottenere un buon equilibrio tra stabilità e leggerezza. L'edizione Alpine, infatti, è particolarmente compatta, il che contribuisce a ridurre i tempi di build e la superficie d'attacco.

All'interno del container, il contesto di lavoro è impostato sulla directory */usr/src/app*, dove vengono copiati i file necessari all'esecuzione del backend. Oltre al file principale *server.js* e ai file di configurazione *package.json*, sono inclusi anche i moduli organizzati in cartelle come

services, graphql, monitors, middlewares e libs, insieme alla directory prisma, che contiene lo schema ORM e le impostazioni per la connessione al database.

La costruzione dell'immagine è pensata per essere efficiente. Le dipendenze vengono installate con l'opzione `--omit=dev`, che esclude i pacchetti utilizzati solo durante lo sviluppo, riducendo il peso finale dell'immagine e eliminando librerie inutili in produzione. Successivamente viene eseguito `npx prisma generate`, che genera automaticamente il client Prisma a partire dallo schema definito, rendendo possibile la comunicazione con il database PostgreSQL.

Per quanto riguarda la sicurezza, il container esegue l'applicazione con un utente non privilegiato (`USER 1000`), evitando l'uso dell'utente root attenendoci alle best practice del principio del minimo privilegio. La porta 4000 viene esposta per accettare le richieste HTTP in ingresso, e il comando finale `CMD ["node", "server.js"]` avvia il server Express all'interno del container.

```
FROM node:lts-alpine3.20
WORKDIR /usr/src/app

COPY package*.json ./
COPY server.js ./
COPY services/ ./services/
COPY prisma/ ./prisma/
COPY monitors/ ./monitors/
COPY middlewares/ ./middlewares/
COPY libs/ ./libs/
COPY graphql/ ./graphql/

RUN npm install --omit=dev
RUN npx prisma generate

USER 1000

EXPOSE 4000
CMD ["node", "server.js"]
```

Figura 2.4: Dockerfile per Express con NodeJs

2.3.2 Keycloak

All'interno dell'infrastruttura, la gestione dell'autenticazione e dell'identità degli utenti è affidata a *Keycloak*, una piattaforma open-source molto diffusa per la gestione centralizzata degli accessi (IAM) in ambienti cloud-native. In questa architettura, Keycloak svolge un ruolo essenziale: agisce come autorità di autenticazione per tutti i servizi applicativi, pur rimanendo nascosto al mondo esterno. I client non interagiscono direttamente con Keycloak, ma passano sempre dal backend Express, che funge da proxy e punto di controllo per tutte le richieste di autenticazione. In questo modo, si ottiene un accesso più sicuro e controllato.

L'istanza Keycloak utilizzata nel progetto è stata estesa tramite *SPI* (*Service Provider Interface*) personalizzati, sviluppati in Java e caricati all'avvio del server. Queste estensioni permettono di adattare il comportamento nativo di Keycloak a scenari più complessi, intervenendo in punti strategici del flusso di autenticazione.

Uno dei moduli più significativi riguarda il controllo condizionale dell'*OTP* (*One-Time Password*). Invece di richiedere sempre il secondo fattore a ogni utente, il sistema verifica se è stata configurata una credenziale OTP attiva. Solo in quel caso, viene richiesto l'inserimento del codice migliorando l'esperienza d'uso per chi non ha attivato la 2FA, senza compromettere la sicurezza di chi invece la utilizza.

Un secondo SPI è stato sviluppato per gestire l'*autenticazione federata*

con *Google* in modalità *direct grant*. Questo modulo consente a client fidati di inviare direttamente un token Google: Keycloak lo valida e, se il processo va a buon fine, collega l'identità federata a un utente esistente o ne crea uno nuovo con le informazioni ricevute. Sono stati implementati anche controlli per evitare conflitti tra email, duplicazioni di profili e garantire la corretta gestione delle azioni richieste all'utente.

A completare il sistema, è stato aggiunto un modulo per l'autenticazione passwordless tramite *WebAuthn*, con endpoint interni dedicati alla registrazione, gestione e verifica delle credenziali FIDO2. Questo SPI estende le funzionalità native di Keycloak, permettendo un'integrazione più personalizzata del protocollo WebAuthn con le logiche applicative del backend.

Infine, l'ecosistema Keycloak è stato arricchito con un *event listener* che tiene sincronizzati i dati utente tra Keycloak e il database dell'applicazione. Ogni volta che un utente viene creato, aggiornato o accede al sistema, l'evento corrispondente viene intercettato e trasmesso direttamente al database PostgreSQL. Questo meccanismo garantisce coerenza tra i dati di autenticazione e quelli dell'applicazione, facilitando audit, tracciamento e personalizzazione dei profili.

Dockerfile

Per distribuire Keycloak all'interno del cluster Kubernetes in modo personalizzato, è stato realizzato un Dockerfile multi-stage basato sull'immagine ufficiale *keycloak:26.2.0*, disponibile su Quay.

Nella prima fase della build, l'immagine opera in modalità builder. Il contesto è temporaneamente elevato a utente *root* per consentire la copia dei file *.jar* contenenti gli SPI personalizzati all'interno della directory *providers/* di Keycloak. Ogni file viene poi assegnato all'utente *keycloak*, con permessi di lettura (644) per evitare problemi in fase di avvio. In questa fase viene utilizzato il comando *touch* per alterare la data di modifica dei file copiati, forzandola in avanti: una soluzione utile per evitare conflitti con il sistema di caching di Keycloak, che in alcune versioni può ignorare provider "più vecchi" rispetto a quelli già noti (bug della versione 26.2.X).

Una volta copiati i file e ripristinato l'utente *keycloak*, viene eseguito il comando *kc.sh build*, che compila la distribuzione di Keycloak includendo gli SPI questo consente di ottimizzare l'immagine ed evitare che le estensioni vengano rielaborate a ogni avvio, migliorando i tempi di startup.

La seconda fase riparte da un'immagine "pulita" della stessa versione di Keycloak. Qui viene montato solo il contenuto necessario prodotto dalla fase di build, escludendo i file temporanei o non indispensabili. In questo modo, l'immagine finale risulta più leggera. Il container viene

poi avviato con il comando `kc.sh start --optimized`, che sfrutta le build precompilate per ridurre ulteriormente il tempo di inizializzazione.

```
FROM quay.io/keycloak/keycloak:26.2.0 AS builder
...
USER root
ENV KC_DB=postgres

ADD --chown=keycloak:keycloak --chmod=644 ./providers/*.jar /opt/keycloak/providers/
RUN touch -m --date=@1743465600 /opt/keycloak/providers/*

USER keycloak
RUN /opt/keycloak/bin/kc.sh build

FROM quay.io/keycloak/keycloak:26.2.0
...
COPY --from=builder /opt/keycloak/ /opt/keycloak/

ENTRYPOINT ["/opt/keycloak/bin/kc.sh", "start", "--optimized"]
```

You, 6 days ago • build

Figura 2.5: Dockerfile per Keycloak 26.2.0

2.3.3 Database

Nel sistema vengono utilizzati due database distinti, entrambi basati su *PostgreSQL*: uno destinato all'applicazione backend (*app-db*) e l'altro riservato a Keycloak (*keycloak-db*). Entrambe le istanze vengono distribuite nel cluster Kubernetes tramite Helm Chart, seguendo un approccio dichiarativo che garantisce coerenza tra le configurazioni e rende più semplice la manutenzione dell'infrastruttura.

Ogni database è definito in un deployment separato, basato sull'immagine ufficiale *postgres:15*, eseguita con una configurazione orientata alla sicurezza: l'applicazione gira come utente non privilegiato, l'accesso con privilegi elevati è disabilitato e il filesystem è impostato in modalità sola lettura. Le credenziali di accesso – come nome del data-

base, utente e password – non sono presenti in chiaro nei manifesti, ma vengono fornite attraverso *Kubernetes Secrets*, montati in modo sicuro grazie all’integrazione con il CSI driver per la gestione dei segreti centralizzando la gestione delle credenziali e riducendo significativamente il rischio di esposizione di dati sensibili.

Dal punto di vista dello storage, ciascun database è associato a un volume persistente dedicato, montato nella directory standard di PostgreSQL. I volumi sono configurati con policy di tipo Retain, così da conservare i dati anche in caso di rimozione accidentale del pod. Per evitare problemi di accesso ai dati, ogni deployment include un init container che si occupa di impostare i corretti permessi sulla directory del volume, assicurando che l’utente PostgreSQL (UID 999) possa accedervi senza restrizioni.

2.4 Comunicazione interna tra i servizi

La comunicazione tra i vari servizi all’interno del cluster è stata progettata per essere controllata e che rispetti i principi delle architetture a microservizi, basando tutto sul modello di deployment su Kubernetes. Ogni componente del sistema — che si tratti del backend applicativo, del database o del sistema di autenticazione — è eseguito all’interno di un *Pod* e reso accessibile internamente tramite un oggetto *Service*, che fornisce un alias DNS stabile e maschera l’indirizzo IP effettivo del pod. Questo, semplifica la risoluzione dei nomi e garantisce una

comunicazione affidabile tra i componenti, anche in caso di scaling o restart.

Il pod express, che ospita il server Node.js dell'applicazione, rappresenta il fulcro della logica applicativa ed è l'unico punto autorizzato a gestire il traffico in entrata nel cluster. Con l'integrazione di un *Ingress Controller NGINX*, tutte le richieste pubbliche — comprese quelle dirette all'API GraphQL e agli endpoint di autenticazione — vengono convogliate attraverso questo nodo.

Nel cluster, il server Express dialoga direttamente con Keycloak sfruttando un service dedicato e utilizzando richieste REST protette. Questo è fondamentale per la validazione dei token JWT, la gestione dei metadati utente e l'orchestrazione di flussi federati o OTP. A sua volta, Keycloak è configurato per accedere esclusivamente a due destinazioni: il backend Express (necessario per completare alcune fasi di autenticazione) e il proprio database, keycloak-db.

Un caso particolare riguarda un modulo SPI personalizzato di Keycloak (l'*EventListenerProvider*), che consente l'invio di eventi direttamente verso app-db. In questo modo, quando avvengono operazioni rilevanti sul profilo utente — come il login o l'aggiornamento dei dati — le informazioni vengono sincronizzate anche con il database dell'applicazione.

Il database app-db è invece accessibile principalmente da Express, che lo utilizza per tutte le operazioni applicative (via Prisma e GraphQL),

e, in modo controllato, da Keycloak per la sincronizzazione dei dati. `keycloak-db`, al contrario, è più isolato: risponde unicamente alle richieste provenienti dal servizio Keycloak stesso.

Sebbene Kubernetes fornisca connettività automatica tra pod tramite i Service, l'infrastruttura prevede l'utilizzo di *NetworkPolicy* per limitare e definire con precisione i flussi di rete ammessi. Ad esempio, si può configurare il cluster affinché solo Express possa accedere a Keycloak, o che `keycloak-db` accetti connessioni esclusivamente dal suo relativo pod. Questo riduce i rischi di accessi non autorizzati e migliora il profilo di sicurezza del sistema.

L'intera rete di comunicazione si basa sulla risoluzione DNS interna di Kubernetes (es. `keycloak.default.svc.cluster.local`), che permette di identificare i servizi in modo chiaro e stabile. La struttura riflette una progettazione attenta, dove ogni componente è definito non solo per le sue funzionalità, ma anche per i limiti entro cui può operare nel sistema.

2.5 Componenti di supporto

Per garantire l'affidabilità, la sicurezza e il monitoraggio dell'intera infrastruttura, sono stati integrati diversi componenti di supporto, ognuno con una funzione specifica e collocato nel proprio namespace. Sebbene non partecipino direttamente all'elaborazione della logica applicativa, questi servizi svolgono un ruolo essenziale nel mantenere

stabile e sicuro l'ambiente in cui l'applicazione opera.

L'installazione dei componenti è avvenuta tramite le rispettive Helm chart, ufficiali o di terze parti, scaricate da repository pubblici o privati. Questa scelta ha reso possibile una configurazione dichiarativa e facilmente versionabile, semplificando la gestione nel tempo e garantendo una netta separazione delle responsabilità tra i vari moduli dell'architettura.

Nelle sezioni successive, vengono analizzati gli strumenti principali adottati per la gestione del reverse proxy, il monitoraggio dei servizi, la gestione dei certificati TLS e l'integrazione sicura dei segreti con Vault.

2.5.1 Nginx

All'interno dell'infrastruttura Kubernetes, il componente NGINX è stato adottato come *Ingress Controller*, con il compito di gestire tutto il traffico HTTP e HTTPS in ingresso. Si tratta del punto di accesso centralizzato dell'applicazione: è qui che confluiscono tutte le richieste provenienti dall'esterno, che poi vengono instradate verso i servizi interni corrispondenti. L'installazione è stata effettuata tramite Helm, utilizzando il chart ufficiale del progetto *ingress-nginx*, e il controller è stato isolato in un namespace dedicato (`ingress-nginx`) semplificando l'organizzazione delle risorse e migliorando la sicurezza del cluster.

Nella fase di configurazione dei values inerenti al chart, sono stati abili-

tati protocolli TLS moderni (1.2 e 1.3), selezionate cipher suite sicure e configurati header HTTP di sicurezza per proteggere le comunicazioni. Inoltre, è stato attivato ModSecurity, integrato con le regole OWASP CRS, per offrire un primo livello di difesa contro attacchi comuni a livello di applicazione. A completare il quadro, è stato implementato un sistema di rate limiting, utile per limitare il numero di richieste per connessione e mitigare eventuali tentativi di attacco DoS.

NGINX è configurato anche per lavorare in sinergia con *Cert-Manager*, da cui riceve dinamicamente i certificati TLS necessari a stabilire connessioni sicure con i client in modo tale che il traffico sia sempre cifrato, garantendo la riservatezza dei dati in transito.

Infine, l'esposizione del controller avviene tramite un servizio di tipo *LoadBalancer*, che consente di rendere disponibili i servizi interni senza esporre direttamente i singoli pod. In questo modo, si mantiene un buon livello di isolamento e si limita la superficie esposta del sistema.

2.5.2 Monitoring

Per garantire un'adeguata supervisione e osservabilità dell'intero cluster Kubernetes e delle applicazioni in esecuzione, è stato adottato il pacchetto Helm *kube-prometheus-stack*, una delle soluzioni più complete e affidabili disponibili in ambito open-source. L'installazione è avvenuta nel namespace dedicato monitoring, utilizzando i repository ufficiali della Prometheus Community e di Grafana, e personalizzando

il comportamento del sistema tramite un file *values.yaml* adattato alle specifiche esigenze del progetto.

Il chart installa e configura un insieme integrato di strumenti: *Prometheus*, per la raccolta delle metriche; *Alertmanager*, per la gestione degli avvisi e delle notifiche; e *Grafana*, per la visualizzazione grafica dei dati raccolti. Oltre ai componenti principali, vengono predisposti automaticamente anche diversi dashboard e oggetti *ServiceMonitor*, utili per il monitoraggio di componenti interni del cluster come *kubelet*, *apiserver*, *controller-manager* e altri.

La configurazione include risorse computazionali dedicate, etichette personalizzate, ingressi HTTP per l'accesso alle interfacce web di *Prometheus* e *Grafana*, e un sistema di autenticazione basilare per proteggere gli endpoint. Inoltre, sono stati configurati *ServiceMonitor* aggiuntivi per estendere il monitoraggio anche a componenti esterni, come ad esempio il controller *NGINX Ingress*, rendendo così possibile tracciare anche l'attività e le performance degli ingressi di rete.

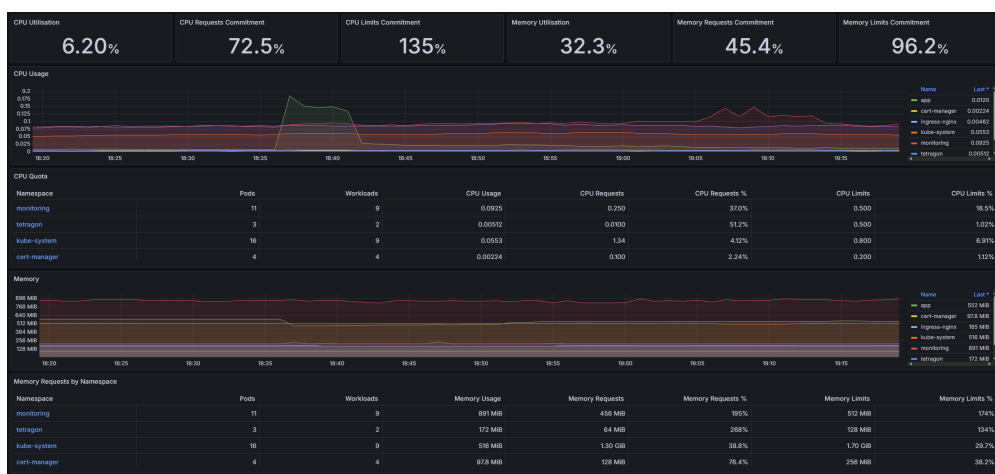


Figura 2.6: Cluster Dashboard in Grafana

2.5.3 Cert Manager

Per la gestione automatizzata dei certificati TLS all'interno del cluster Kubernetes, è stato adottato *Cert-Manager*, installato tramite il chart ufficiale rilasciato da *Jetstack*. Il componente è stato distribuito nel namespace dedicato `cert-manager` e configurato attraverso un file *values.yaml* personalizzato, così da assicurare un'integrazione fluida con gli altri servizi e un livello di sicurezza adeguato.

Cert-Manager si occupa della creazione, rinnovo e revoca dei certificati, sfruttando risorse Kubernetes come *Certificate*, *Issuer* e *ClusterIssuer*. La configurazione implementata nel progetto include anche il supporto per l'esposizione delle metriche tramite Prometheus, grazie all'attivazione dei ServiceMonitor, e l'opzione `enableCertificateOwnerRef`, che semplifica la gestione dei secret associati ai certificati rendendola più automatica e affidabile.

Dal punto di vista della sicurezza, il componente è stato impostato per funzionare con container non privilegiati, limiti sulle risorse, policy restrittive e un webhook che valida i certificati in ingresso e gestisce la comunicazione con i controller. È stato inoltre abilitato *cainjector*, il sottocomponente incaricato di inserire automaticamente i certificati nei workload che ne hanno bisogno, e configurato lo *startupapicheck*, un meccanismo che verifica l'avvio corretto del webhook all'installazione.

2.5.4 Vault e Vault CSI

Per la gestione sicura di segreti e credenziali all'interno del cluster Kubernetes, è stato adottato *HashiCorp Vault*, installato tramite il chart ufficiale Helm e isolato nel namespace dedicato *vault*. Vault funge da secret store centralizzato, responsabile del salvataggio, della gestione e della distribuzione di informazioni sensibili come token di accesso, password di database, certificati e chiavi API.

A supporto di Vault, è stato installato anche il *driver CSI (Container Storage Interface)* nella sua implementazione specifica per Kubernetes, *secrets-store-csi-driver*, posizionato nel namespace *kube-system*. Questo permette ai pod di montare i segreti direttamente come file all'interno di volumi temporanei, evitando il passaggio tramite risorse Kubernetes tradizionali come ConfigMap o Secret.

L'integrazione tra Vault e il driver CSI è stata realizzata attraverso la risorsa *SecretProviderClass*, che definisce quali segreti Vault deve esporre e a quali pod. Grazie a questo meccanismo, le applicazioni possono accedere in modo sicuro e puntuale solo alle credenziali di cui hanno bisogno, senza che queste vengano mai salvate in chiaro nel cluster.

Vault e il driver CSI operano in stretta sinergia: il primo conserva i dati sensibili e applica le policy di accesso, il secondo li rende disponibili solo ai pod autorizzati, montandoli come file in sola lettura eliminando la necessità di gestire segreti statici nel cluster.

Capitolo 3

Sicurezza

dell'Infrastruttura

Questo capitolo è dedicato all'analisi delle principali strategie adottate per garantire la sicurezza dell'infrastruttura Kubernetes su cui si basa l'intero sistema. In un ambiente distribuito come Kubernetes, dove numerosi pod, servizi e componenti comunicano tra loro e con l'esterno, la protezione dell'infrastruttura rappresenta un aspetto cruciale. Per rispondere a questa esigenza, sono state implementate diverse misure tecniche e configurazioni mirate, che verranno approfondite nelle sezioni seguenti.

Si partirà dall'utilizzo dei *SecurityContext* nei deployment, che permettono di impostare regole di esecuzione sicura per i container, come l'esecuzione con utenti non privilegiati, l'impossibilità di ottenere permessi elevati e l'uso di filesystem in sola lettura. Queste impostazioni

rappresentano il primo strato di difesa all'interno dei pod.

A seguire, si analizzerà il ruolo delle *NetworkPolicy*, strumenti essenziali per limitare e controllare il traffico tra i pod, riducendo la superficie d'attacco interna al cluster e rendendo più prevedibili i flussi di rete.

Un'altra area chiave riguarda la configurazione dell'*Ingress Controller NGINX*, dove sono stati applicati specifici parametri di sicurezza — come il supporto a protocolli TLS moderni, l'impostazione di header protettivi e la gestione dei limiti di connessione — per proteggere il traffico HTTP/S da attacchi comuni e garantire un accesso sicuro ai servizi esposti.

La gestione dei segreti è stata invece affidata a *Vault*, integrato con il *driver CSI*, che consente ai pod di accedere in modo sicuro ai segreti montati come file temporanei. Questo approccio elimina la necessità di utilizzare Secret statici di Kubernetes e consente un controllo centralizzato e granulare delle credenziali sensibili. Infine, verrà illustrata l'integrazione di *Cert-Manager*, il componente incaricato di emettere e gestire automaticamente i certificati TLS all'interno del cluster.

3.1 SecurityContext nei deployment

All'interno di un cluster Kubernetes, il *SecurityContext* è uno strumento essenziale per definire il comportamento di sicurezza dei container in esecuzione. Tramite questa configurazione è possibile controllare vari

aspetti legati ai privilegi, agli utenti utilizzati nei processi, all'accesso al filesystem e all'uso di capacità del kernel. L'obiettivo è limitare il più possibile i permessi dei container, applicando il principio del minimo privilegio per ridurre i rischi in caso di compromissione.

3.1.1 Express

Nel deployment del servizio Express, il container viene eseguito con l'*UID 1000*, in modalità non-root, grazie all'impostazione *runAsNonRoot: true*. Per rafforzare ulteriormente la sicurezza, è stata disabilitata ogni forma di escalation dei privilegi (*allowPrivilegeEscalation: false*) e il filesystem è stato montato in sola lettura, così da evitare modifiche non controllate. Inoltre, tutte le capability Linux sono state rimosse (*capabilities.drop: ["ALL"]*), riducendo al minimo la superficie d'attacco del processo. Queste scelte contribuiscono a rendere il container più sicuro e resistente a eventuali exploit.

```
securityContext:
  runAsUser: 1000
  runAsNonRoot: true
  allowPrivilegeEscalation: false
  readOnlyRootFilesystem: true
  capabilities:
    drop: ["ALL"]
```

Figura 3.1: SecurityContext nel Deployment di Express

3.1.2 Keycloak

Anche il pod di Keycloak segue una configurazione simile. Il container viene avviato con lo stesso *UID* (1000), senza privilegi elevati e con il filesystem impostato in modalità *read-only*. Le capability di sistema sono state completamente eliminate, creando un ambiente più isolato e sicuro. Questo tipo di configurazione è particolarmente importante, considerando che Keycloak gestisce l'autenticazione e l'autorizzazione di tutti gli utenti del sistema, ricoprendo quindi un ruolo altamente sensibile.

```
securityContext:
  runAsUser: 1000
  runAsNonRoot: true
  allowPrivilegeEscalation: false
  readOnlyRootFilesystem: true
  capabilities:
    drop: ["ALL"]
```

Figura 3.2: SecurityContext nel Deployment di Keycloak

3.1.3 Database

Sia il database di Keycloak che il database applicativo (app-db), entrambi basati su PostgreSQL, adottano una configurazione di sicurezza coerente con le best practice dell'immagine ufficiale. In entrambi i deployment, i container vengono eseguiti con l'*UID* 999, in modalità *non-root*, con privilegi elevati disattivati e filesystem montato in sola lettura. Anche in questi casi, tutte le capability Linux sono state

rimosse, garantendo un contesto di esecuzione estremamente limitato e controllato permettendo di proteggere adeguatamente i componenti responsabili della persistenza dei dati e mantenendo elevati standard di hardening.

```
securityContext:  
  runAsUser: 999  
  runAsGroup: 999  
  runAsNonRoot: true  
  allowPrivilegeEscalation: false  
  readOnlyRootFilesystem: true  
  capabilities:  
    drop: ["ALL"]
```

Figura 3.3: SecurityContext nei Deployment dei Database

3.2 NetworkPolicy applicate ai pod

Nel contesto di Kubernetes, le NetworkPolicy rappresentano uno strumento essenziale per regolare il traffico di rete all'interno del cluster. Consentono di definire con precisione quali comunicazioni sono permesse tra i pod e verso l'esterno, introducendo una forma di firewall interno. In assenza di policy specifiche, Kubernetes consente tutto il traffico in modo predefinito; per questo motivo, adottare una strategia restrittiva è una buona pratica per ridurre la superficie d'attacco e isolare i componenti tra loro.

All'interno di questa infrastruttura è stato adottato un approccio “deny by default”, bloccando ogni connessione in ingresso e in uscita tramite

una regola globale (*default-deny-all*), e abilitando solo i flussi strettamente necessari. Di seguito vengono descritte le policy applicate ai principali pod.

3.2.1 Default-Deny-All

Il primo livello di protezione è costituito da una regola che nega tutto il traffico in entrata e in uscita per ogni pod, salvo quanto esplicitamente consentito da altre policy. Questo garantisce un comportamento prevedibile e chiaro, impedendo comunicazioni non autorizzate tra i workload.

```
#DENY ALL INGRESS/EGRESS
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

Figura 3.4: Network Policy Default-Deny-All

3.2.2 Accesso da Prometheus e Grafana

Per abilitare il monitoraggio delle metriche da parte degli strumenti di osservabilità, è stata definita una policy che consente ai pod di Prometheus, Grafana, Node Exporter e altri componenti presenti nel

namespace monitoring di accedere ai pod nel namespace app. In questo modo è possibile raccogliere dati sulle performance e lo stato del sistema senza aprire canali superflui.

```
#ALLOW PROMETHEUS/GRAPHANA METRICS
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-prometheus-scraper
  namespace: app
spec:
  podSelector: {}
  ingress:
    - from:
      - namespaceSelector:
          matchLabels:
            name: monitoring
        podSelector:
          matchLabels:
            app.kubernetes.io/name: grafana
      - namespaceSelector:
          matchLabels:
            name: monitoring
        podSelector:
          matchLabels:
            app.kubernetes.io/name: prometheus
      - namespaceSelector:
          matchLabels:
            name: monitoring
        podSelector:
          matchLabels:
            app.kubernetes.io/name: prometheus-node-exporter
      - namespaceSelector:
          matchLabels:
            name: monitoring
```

Figura 3.5: Estratto della Network Policy per il monitoring

3.2.3 Express

Il servizio Express è configurato per ricevere traffico in ingresso esclusivamente dall'Ingress Controller NGINX, assicurando che tutte le richieste passino per il punto di accesso previsto. In uscita, Express può comunicare con Keycloak (per l'autenticazione), con app-db (per la persistenza dei dati), con i server DNS per la risoluzione dei nomi e, quando necessario, con servizi esterni via HTTPS (porta 443).

```
# EXPRESS
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: express-policy
  namespace: app
spec:
  podSelector:
    matchLabels:
      app: express
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              name: ingress-nginx
          podSelector:
            matchLabels:
              app.kubernetes.io/name: ingress-nginx
              app.kubernetes.io/component: controller
  egress:
    - to:
        - podSelector:
            matchLabels:
              app: keycloak
        - podSelector:
            matchLabels:
              app: app-db
    - to:
        - namespaceSelector: {}
      ports:
        - protocol: UDP
          port: 53
      # DNS
    - to:
        - ipBlock:
            cidr: 0.0.0.0/0
      ports:
        - protocol: TCP
          port: 443
      # Internet
  policyTypes:
    - Ingress
    - Egress
```

Figura 3.6: Network Policy per Express

3.2.4 Keycloak

Anche il pod Keycloak è protetto da una policy che consente connessioni in ingresso solo da NGINX e da Express, che lo interroga per gestire i flussi di login e validazione. In uscita, può comunicare esclusivamente con i database keycloak-db e app-db, oltre che con il servizio DNS. Si limitano così al minimo i percorsi di comunicazione, mantenendo il componente isolato. Per l'esposizione del pannello di controllo anche il pod Keycloak interagisce con l'ingress NGINX.

```
# KEYCLOAK
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: keycloak-policy
  namespace: app
spec:
  podSelector:
    matchLabels:
      app: keycloak
  ingress:
    - from:
      - namespaceSelector:
          matchLabels:
            name: ingress-nginx
        podSelector:
          matchLabels:
            app.kubernetes.io/name: ingress-nginx
            app.kubernetes.io/component: controller
      - podSelector:
          matchLabels:
            app: express
    egress:
      - to:
        - podSelector:
            matchLabels:
              app: app-db
        - podSelector:
            matchLabels:
              app: keycloak-db
      - to:
        - namespaceSelector: {} # DNS
        ports:
          - protocol: UDP
            port: 53
  policyTypes:
    - Ingress
    - Egress
```

Figura 3.7: Network Policy per Keycloak

3.2.5 Database keycloak-db

Il database di Keycloak è configurato per ricevere connessioni unicamente dal pod Keycloak stesso. Anche in uscita, le comunicazioni sono limitate allo stesso Keycloak e ai server DNS. In questo modo, il database resta completamente isolato da qualsiasi altra componente dell'infrastruttura.

```
# KEYCLOAK-DB
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: keycloak-db-policy
  namespace: app
spec:
  podSelector:
    matchLabels:
      app: keycloak-db
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: keycloak
  egress:
    - to:
        - podSelector:
            matchLabels:
              app: keycloak
    - to:
        - namespaceSelector: {} # DNS
  ports:
    - protocol: UDP
      port: 53
  policyTypes:
    - Ingress
    - Egress
```

Figura 3.8: Network Policy per il Database di Keycloak

3.2.6 Database app-db

Il database applicativo (app-db) accetta connessioni solo da Express e Keycloak, che sono i servizi autorizzati ad accedere ai dati. Le risposte alle query e le richieste DNS costituiscono le uniche comunicazioni in uscita per questo pod, mantenendo alta la compartimentazione.

```
# APP-DB
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: app-db-policy
  namespace: app
spec:
  podSelector:
    matchLabels:
      app: app-db
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: express
        - podSelector:
            matchLabels:
              app: keycloak
  egress:
    - to:
        - podSelector:
            matchLabels:
              app: express
        - podSelector:
            matchLabels:
              app: keycloak
    - to:
        - namespaceSelector: {} # DNS
  ports:
    - protocol: UDP
      port: 53
  policyTypes:
    - Ingress
    - Egress
```

Figura 3.9: Network Policy per il Database dell'app

3.3 Configurazione dell'Ingress Controller

All'interno dell'infrastruttura Kubernetes, il componente incaricato di gestire il traffico in ingresso è NGINX Ingress Controller, scelto per la sua flessibilità e affidabilità. L'installazione è avvenuta tramite Helm, utilizzando il chart ufficiale, e configurata in modo da garantire un elevato livello di sicurezza nel trattamento del traffico HTTP e HTTPS.

3.3.1 Configurazione di NGINX

La configurazione del controller è stata personalizzata direttamente nel file *nginx-values.yaml*, con particolare attenzione a proteggere l'applicazione da abusi e attacchi comuni. I parametri principali adottati includono:

- **Limitazione del traffico:** impostazioni come *limit-connections*, *limit-rpm* e *limit-rps* servono a contenere il numero di richieste che ogni client può effettuare, limitando così i potenziali impatti di attacchi DoS.
- **Controllo delle dimensioni delle richieste:** parametri come *large-client-header-buffers*, *client-body-buffer-size* e *client-max-body-size* definiscono i limiti massimi per header e payload, contribuendo a prevenire attacchi come buffer overflow o upload eccessivi.
- **Protezione da vulnerabilità applicative:** l'integrazione con ModSecurity e le regole OWASP CRS è stata abilitata tramite *enable-modsecurity* e *enable-owasp-modsecurity-crs*, trasformando il controller in un vero e proprio WAF (Web Application Firewall) capace di bloccare minacce come SQL injection, XSS e path traversal.
- **Header di sicurezza:** impostazioni come *server-tokens*: false (per non esporre la versione di NGINX), e opzioni per HSTS

(*hsts*, *hsts-max-age*, *hsts-include-subdomains*, *hsts-preload*) rafforzano la protezione contro intercettazioni, attacchi man-in-the-middle e tecniche di fingerprinting.

- **Gestione degli indirizzi IP reali:** grazie ai parametri *use-forwarded-headers*, *compute-full-forwarded-for* e *real-ip-header*, NGINX è in grado di risalire correttamente all'indirizzo IP del client anche in presenza di proxy o bilanciatori, garantendo una tracciabilità più accurata.

3.3.2 Ingress per il Servizio Express

Per esporre il servizio Express all'esterno del cluster è stato definito un oggetto Ingress, configurato per operare esclusivamente in HTTPS. L'oggetto include diverse annotazioni pensate per rafforzare la sicurezza della comunicazione:

- Le direttive *nginx.ingress.kubernetes.io/ssl-redirect: "true"* e *force-ssl-redirect: "true"* assicurano che ogni richiesta HTTP venga automaticamente reindirizzata su HTTPS.
- Le opzioni *secure-backends: "true"* e *backend-protocol: "HTTP"* migliorano la protezione nel transito dei dati tra il controller e i pod interni.

- L'annotazione *proxy-body-size: "10m"* impone un limite alle dimensioni dei payload accettati, coerente con le impostazioni globali di NGINX.

Nel blocco `tls` è stato specificato l'utilizzo di un certificato unificato (*unified-cert*), valido per tutti gli host dell'applicazione. Il dominio `api.frascoengineer.com` è associato al servizio `Express`, esposto sulla path `/` e instradato verso la porta 80 del service interno.

3.3.3 Accesso sicuro ai pannelli admin

In un'infrastruttura *Kubernetes* esposta su Internet, la protezione dell'accesso ai pannelli di amministrazione – come *Keycloak*, *Vault* o *Grafana* – rappresenta una priorità. Questi servizi, se lasciati pubblicamente accessibili, possono diventare bersagli di attacchi automatizzati o mirati.

L'obiettivo è garantire che solo noi possiamo accedere da remoto a tali pannelli, non avendo però a disposizione un IP statico (tipico delle connessioni domestiche). Di conseguenza, è stato necessario implementare un sistema flessibile in grado di aggiornare dinamicamente i criteri di accesso.

La soluzione implementata è stata quella di usare un *DDNS* con aggiornamento dinamico degli *Ingress*:

1. Hostname dinamico con No-IP

Per superare il limite dell'IP dinamico, è stato utilizzato il servi-

zio *No-IP*, che associa un nome DNS stabile (`nsproject.ddns.net`) al nostro IP pubblico, anche quando questo cambia. In questo modo, è sempre possibile risolvere l'IP corrente tramite DNS.

2. Ingress NGINX e whitelist IP

Gli *Ingress Controller* di tipo *NGINX* supportano un'annotazione specifica che consente di filtrare il traffico in ingresso in base all'indirizzo IP sorgente:

```
nginx.ingress.kubernetes.io/whitelist-source-range "X.X.X.X/32"
```

Questa annotazione viene letta dinamicamente e applicata al traffico *HTTP/HTTPS* diretto verso il servizio, consentendo di limitare l'accesso a un solo IP o a una rete specifica.

3. Aggiornamento automatico con CronJob Kubernetes

Per automatizzare tutto il processo, è stato implementato un *CronJob* che:

- ogni 5 minuti risolve l'IP corrente del dominio `nsproject.ddns.net`
- aggiorna dinamicamente gli ingressi dei pannelli di amministrazione (*Keycloak*, *Vault*, *Grafana*),
- applica l'IP ottenuto alla whitelist come `X.X.X.X/32`, consentendo esclusivamente il nostro accesso.

Questo sistema simula un comportamento da IP statico pur utilizzando una connessione dinamica.

```
command:
  - /bin/sh
  - -c
  - |
    IP=$(nslookup nsproject.ddns.net | awk '/^Address: / { print $2 }' | tail -n1)
    echo "$IP" > /data/ip.txt
volumeMounts:
  - name: shared-data
    mountPath: /data
name: patcher
image: bitnami/kubectl:latest
command:
  - /bin/sh
  - -c
  - |
    sleep 10
    IP=$(cat /data/ip.txt)
    if [ -z "$IP" ]; then
      echo "Errore: IP non trovato"
      exit 1
    fi
    echo "Patching con IP: $IP"
    kubectl patch ingress monitoring-nginx-ingress \
      -n monitoring \
      --type=merge \
      -p "{\"metadata\":{\"annotations\":{\"nginx.ingress.kubernetes.io/whitelist-source-range\":\"$IP/32\"}}}"
```

Figura 3.10: CronJob per l'aggiornamento degli IP

3.4 Gestione dei segreti con Vault e Vault-CSI

Per assicurare una gestione sicura e centralizzata delle credenziali sensibili, l'infrastruttura fa uso di *HashiCorp Vault*, integrato con il *Vault CSI Provider*. Questo approccio permette ai pod di accedere ai segreti in modo dinamico, evitando di salvarli staticamente nei manifest Kubernetes e riducendo così i rischi legati all'esposizione accidentale delle

credenziali.

3.4.1 Accesso controllato tramite ServiceAccount

Ogni pod che necessita di accedere a segreti gestiti da Vault è associato a un *ServiceAccount* specifico. Questo account è collegato a un ruolo definito in Vault, il quale, a sua volta, è autorizzato a leggere determinati percorsi (*secret path*) attraverso una o più policy. In pratica, componenti come Express, Keycloak e i due database utilizzano ciascuno un ServiceAccount dedicato (es. `express-sa`, `keycloak-sa`, `app-db-sa`), con il parametro `automountServiceAccountToken: false` attivato per limitare l'accesso implicito ai token di autenticazione, riducendo la superficie di attacco.

La mappatura tra ServiceAccount e ruoli Vault avviene tramite il *Vault Kubernetes Auth Backend*, che verifica il token del ServiceAccount e applica le autorizzazioni definite, consentendo così un controllo preciso e affidabile sugli accessi.

3.4.2 SecretProviderClass (SPC)

Il collegamento operativo tra Kubernetes e Vault viene gestito attraverso gli oggetti *SecretProviderClass* (SPC). Si tratta di risorse personalizzate che descrivono:

- Il provider usato (in questo caso vault),

- L'indirizzo del server Vault,
- Il ruolo da utilizzare per l'autenticazione (`roleName`),
- E una lista di segreti da recuperare, con indicazione del percorso (`secretPath`), della chiave (`secretKey`) e del nome con cui il dato verrà esposto nel container (`objectName`).

Con questa configurazione, il pod può accedere ai segreti montati direttamente come file in sola lettura, disponibili solo durante il runtime e non persistenti nel cluster.

3.4.3 `syncSecret`

Una funzionalità utile offerta dal CSI driver è *syncSecret*, configurabile tramite il campo `secretObjects` nello SPC. Quando attiva, questa opzione permette di generare automaticamente un Secret Kubernetes a partire dai segreti ottenuti da Vault. Questo è particolarmente vantaggioso per quelle applicazioni che richiedono l'uso di *env.valueFrom.secretKeyRef*, come nel caso di PostgreSQL o Keycloak.

Ad esempio, per il database *app-db*, i segreti montati da Vault vengono trasformati in un oggetto Secret chiamato *app-db-k8s-secret*, che può poi essere referenziato nei container tramite variabili d'ambiente bilanciando l'accessibilità dei dati sensibili con il rispetto dei criteri di sicurezza.

3.5 Certificati SSL tramite Cert-Manager

Per garantire che la comunicazione tra client esterni e servizi esposti sia sempre protetta, l'infrastruttura utilizza Cert-Manager, uno strumento pensato per gestire in modo automatico l'emissione e il rinnovo dei certificati TLS all'interno di ambienti Kubernetes. Il componente si integra perfettamente con il protocollo ACME (utilizzato da Let's Encrypt) e supporta anche provider DNS come Cloudflare per la validazione dei domini.

3.5.1 ClusterIssuer e Certificate

Alla base del funzionamento di Cert-Manager ci sono due risorse principali:

- Il *ClusterIssuer*, che definisce a livello di cluster come e da dove ottenere i certificati. In questo caso, l'autorità configurata è Let's Encrypt, con endpoint ACME e una chiave privata archiviata in un secret, usata per autenticare le richieste.
- Il *Certificate*, che specifica i dettagli del certificato da emettere, come i domini da proteggere, la durata del certificato e il tempo di rinnovo. Una volta approvata la richiesta, Cert-Manager crea automaticamente un Secret contenente il certificato e la relativa chiave privata.

Nel progetto, il certificato denominato `unified-cert` è stato configurato per coprire tutti i domini necessari per l'esposizione delle api e dei pannelli di amministrazione.

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-cloudflare
spec:
  acme:
    email: f[REDACTED]@hotmail.it
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: letsencrypt-cloudflare-key
    solvers:
      - dns01:
          cloudflare:
            apiTokenSecretRef:
              name: cloudflare-api-token-secret
              key: api-token
```

Figura 3.11: ClusterIssuer

3.5.2 Integrazione con Vault per le credenziali Cloudflare

Un aspetto rilevante della configurazione è l'integrazione con Vault per la gestione sicura del token API di Cloudflare, utilizzato da Cert-Manager per la validazione dei domini via DNS. Il token non è inserito direttamente nei manifesti, ma viene recuperato dinamicamente attraverso il Vault CSI driver.

A tal fine, è stato definito uno `SecretProviderClass` (`cloudflare-certmanager`) che permette di montare il token come file all'interno del pod. Per ren-

dere questo token disponibile anche come Secret Kubernetes, è stato creato un Job (`sync-cloudflare-secret`) che effettua la sincronizzazione, garantendo così a Cert-Manager l'accesso alle credenziali durante il processo di validazione DNS.

Capitolo 4

SecDevOps

Questo capitolo approfondisce come la sicurezza sia stata integrata in modo sistematico all'interno del ciclo di sviluppo e rilascio del software, seguendo un approccio *SecDevOps*. A differenza del modello DevOps tradizionale, che pone l'accento sulla velocità e sull'automazione dei processi, il paradigma SecDevOps mira a inserire la sicurezza come elemento strutturale in ogni fase del ciclo: dalla scrittura del codice fino alla messa in produzione.

Viene illustrata una pipeline CI/CD realizzata con GitLab, progettata per automatizzare controlli di qualità e sicurezza sia sul codice che sull'infrastruttura. Il flusso è suddiviso in diversi stage che includono:

- l'analisi statica del codice (SAST),
- la scansione delle immagini Docker per individuare vulnerabilità note,

- la costruzione e pubblicazione delle immagini in un registry dedicato,
- la validazione dei manifesti Kubernetes,
- il deploy automatico nel cluster,
- il controllo della sicurezza dell'ambiente di runtime,
- test dinamici (DAST) eseguiti sull'applicazione in esecuzione.

Particolare attenzione è dedicata alla raccolta e aggregazione dei report generati da ogni fase, che vengono consolidati per offrire una visione centralizzata e chiara dello stato di sicurezza del progetto. A completare il processo, viene descritto il meccanismo di mitigazione delle vulnerabilità, con esempi concreti su come individuare gli errori, intervenire sul codice o sull'infrastruttura, e rilanciare la pipeline fino al raggiungimento di uno stato conforme.

4.1 Cos'è il SecDevOps?

SecDevOps rappresenta l'evoluzione naturale del modello DevOps, con l'obiettivo di integrare la sicurezza come parte integrante del ciclo di vita del software, e non come un'attività separata o finale. Se DevOps punta a velocizzare lo sviluppo attraverso automazione e rilascio continuo, il modello SecDevOps estende questi principi, includendo fin da

subito pratiche orientate alla protezione del codice e dell'infrastruttura.

In questa prospettiva, attività come l'analisi statica del codice, la scansione delle immagini container, la validazione delle configurazioni infrastrutturali e i test di sicurezza runtime vengono completamente automatizzate e rese parte integrante della pipeline CI/CD. Questo consente di intercettare vulnerabilità in fase precoce, riducendo tempi e costi di intervento e rendendo il processo di rilascio più solido.

Nel progetto descritto, l'approccio SecDevOps è stato applicato attraverso una pipeline GitLab CI/CD, strutturata per eseguire una serie di controlli automatizzati: analisi statica del codice (SAST), scansioni container con Trivy, validazione dei manifest e chart Helm, analisi della sicurezza del cluster Kubernetes tramite Kubescape e test dinamici sull'applicazione in esecuzione (ZAP DAST).

4.2 Pipeline GitLab CI/CD

La pipeline GitLab CI/CD implementata in questo progetto integra controlli di sicurezza automatizzati lungo tutto il ciclo di vita del software. È suddivisa in più stage sequenziali, ciascuno dedicato ad attività specifiche come l'analisi statica del codice, la scansione delle immagini container, la validazione dei manifesti Kubernetes, il deployment, l'analisi della sicurezza del cluster e i test dinamici sull'applicazione.

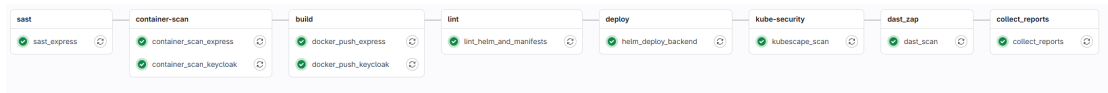


Figura 4.1: Pipeline in Gitlab

4.2.1 Analisi statica del codice (SAST)

Il primo stadio della pipeline è dedicato all'analisi statica del codice sorgente (*SAST*, *Static Application Security Testing*), con l'obiettivo di individuare vulnerabilità e problemi di sicurezza direttamente all'interno del codice, prima ancora che venga eseguito o distribuito.

In questa fase, l'attenzione è rivolta in particolare all'applicazione Express, la cui struttura viene analizzata alla ricerca di potenziali criticità tramite due strumenti principali:

- **ESLint**: un linter per JavaScript che consente di rilevare errori sintattici, pattern di codice sospetti e pratiche non conformi. All'interno della pipeline, viene eseguito con l'opzione `-f json`, generando un report (*eslint_report.json*) in formato JSON, facilmente riutilizzabile per successive elaborazioni automatiche.
- **NodeJsScan**: uno strumento pensato appositamente per l'analisi statica di applicazioni sviluppate in Node.js. È in grado di individuare vulnerabilità legate alla sicurezza, come XSS, SQL injection o l'uso di moduli insicuri, oltre a eventuali configurazioni non corrette. Anche in questo caso, il risultato dell'analisi è un report JSON (*nodejsscan_report.json*).

Durante l'esecuzione di questo stage, entrambi gli strumenti analizzano il contenuto della directory *kubernetes/dockers/express*. I report generati vengono salvati come artifact, così da poter essere consultati negli stadi successivi e integrati nel report finale aggregato.

```
sast_express:
  stage: sast
  image: python:3.11-alpine
  before_script:
    - apk add --no-cache git nodejs npm
    - pip install nodejsscan
    - npm install -g eslint
  script:
    - cd kubernetes/dockers/express
    - eslint . -f json -o "$CI_PROJECT_DIR/eslint_report.json"
    - nodejsscan -d . -o "$CI_PROJECT_DIR/nodejsscan_report.json"
    - |
      python3 -c "
      import json, sys
      with open('$CI_PROJECT_DIR/nodejsscan_report.json') as f:
        report = json.load(f)
        total = report.get('total_count', {})
        if total.get('sec', 0) > 0 or total.get('mis', 0) > 0:
          print('Security issues found: failing job.')
          sys.exit(1)
      "
  artifacts:
    when: always
    paths:
      - eslint_report.json
      - nodejsscan_report.json
  only:
    - push
```

Figura 4.2: Job sast_express

4.2.2 Container Scanning

Dopo l'analisi statica del codice, la pipeline prosegue con la scansione delle immagini Docker, una fase essenziale per individuare vulnerabilità note all'interno dei pacchetti di sistema e delle dipendenze applica-

tive. Questo controllo è fondamentale per assicurarsi che le immagini container utilizzate non contengano componenti a rischio, riducendo così le potenziali superfici d'attacco.

Lo strumento scelto per questa attività è:

- **Trivy:** un tool open-source sviluppato da *Aqua Security*, pensato appositamente per la scansione di immagini container. È in grado di rilevare *CVE* (*Common Vulnerabilities and Exposures*) di livello HIGH e CRITICAL, analizzando sia i pacchetti del sistema operativo (ad esempio Alpine, Debian), sia le dipendenze applicative (come quelle gestite da npm o pip).

Nella pipeline sono stati definiti due job distinti: uno dedicato all'immagine dell'applicazione Express e l'altro all'immagine di Keycloak, entrambi localizzati nella directory *kubernetes/dockers/*. Ogni job prevede la build dell'immagine Docker seguita dalla scansione con Trivy, che produce un report in formato JSON (*trivy_report_express.json* e *trivy_report_keycloak.json*), poi salvato come artifact per l'aggregazione nel report finale.

```
container_scan_express:
  stage: container-scan
  image: docker:latest
  services:
    - docker:dind
  variables:
    DOCKER_HOST: tcp://docker:2375/
    DOCKER_TLS_CERTDIR: ""
  before_script:
    - apk add --no-cache curl
    - curl -sFL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sh -s -- -b /usr/local/bin
  script:
    - cd kubernetes/dockers/express
    - docker build -t express-backend:ci .
    - trivy image --exit-code 0 --severity HIGH,CRITICAL --format json -o trivy_report_express.json express-backend:ci
    - trivy image --severity HIGH,CRITICAL express-backend:ci --exit-code 1
  artifacts:
    when: always
    paths:
      - kubernetes/dockers/express/trivy_report_express.json
  needs:
    - job: sast_express
    artifacts: false
  only:
    - push
```

Figura 4.3: Job container_scan_express

```
container_scan_keycloak:
  stage: container-scan
  image: docker:latest
  services:
    - docker:dind
  variables:
    DOCKER_HOST: tcp://docker:2375/
    DOCKER_TLS_CERTDIR: ""
  before_script:
    - apk add --no-cache curl
    - curl -sFL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sh -s -- -b /usr/local/bin
  script:
    - cd kubernetes/dockers/keycloak
    - docker build -t keycloak-custom:ci .
    - trivy image --exit-code 0 --severity HIGH,CRITICAL --format json -o trivy_report_keycloak.json keycloak-custom:ci
    - trivy image --severity HIGH,CRITICAL keycloak-custom:ci --exit-code 1
  artifacts:
    when: always
    paths:
      - kubernetes/dockers/keycloak/trivy_report_keycloak.json
  only:
    - push
```

Figura 4.4: Job container_scan_keycloak

4.2.3 Build e Push

Dopo aver completato le fasi di analisi statica e scansione delle immagini, la pipeline procede con la creazione e pubblicazione delle immagini Docker aggiornate, pronte per il deploy garantendo tracciabilità e coe-

renza tra il codice sorgente e gli ambienti di esecuzione.

I due job principali coinvolti in questa fase sono:

- *docker_push_express*
- *docker_push_keycloak*

Entrambi utilizzano Docker-in-Docker (dind) per eseguire il build a partire dai Dockerfile presenti nelle directory *kubernetes/dockers/express* e *kubernetes/dockers/keycloak*.

Ogni immagine viene taggata in due modalità:

- latest, per ambienti di sviluppo o test;
- SHA del commit corrente (\$CI_COMMIT_SHORT_SHA), per creare una versione univoca e tracciabile, ideale nei flussi CI/CD.

Una volta completata la build, le immagini vengono pubblicate nel *GitLab Container Registry*, utilizzando il *CI_JOB_TOKEN* per l'autenticazione. Questa fase è eseguita solo se le scansioni di sicurezza precedenti sono andate a buon fine, assicurando che vengano distribuite solo immagini verificate e sicure, pronte per il deploy nei successivi stadi della pipeline.

```
docker_push_express:
  stage: build
  image: docker:latest
  services:
    - docker:dind
  variables:
    DOCKER_HOST: tcp://docker:2375/
    DOCKER_TLS_CERTDIR: ""
  before_script:
    - echo "$CI_JOB_TOKEN" | docker login -u "$CI_REGISTRY_USER" --password-stdin "$CI_REGISTRY"
  script:
    - cd kubernetes/docker/express
    - docker build -t "$CI_REGISTRY_IMAGE/express:latest" -t "$CI_REGISTRY_IMAGE/express:$CI_COMMIT_SHORT_SHA" .
    - docker push "$CI_REGISTRY_IMAGE/express:$CI_COMMIT_SHORT_SHA"
  needs:
    - job: container_scan_express
      artifacts: false
    - job: container_scan_keycloak
      artifacts: false
  only:
    - push
```

Figura 4.5: Job docker_push_express

```
docker_push_keycloak:
  stage: build
  image: docker:latest
  services:
    - docker:dind
  variables:
    DOCKER_HOST: tcp://docker:2375/
    DOCKER_TLS_CERTDIR: ""
  before_script:
    - echo "$CI_JOB_TOKEN" | docker login -u "$CI_REGISTRY_USER" --password-stdin "$CI_REGISTRY"
  script:
    - cd kubernetes/docker/keycloak
    - docker build -t "$CI_REGISTRY_IMAGE/keycloak:latest" -t "$CI_REGISTRY_IMAGE/keycloak:$CI_COMMIT_SHORT_SHA" .
    - docker push "$CI_REGISTRY_IMAGE/keycloak:$CI_COMMIT_SHORT_SHA"
  needs:
    - job: container_scan_keycloak
      artifacts: false
    - job: container_scan_express
      artifacts: false
  only:
    - push
```

Figura 4.6: Job docker_push_keycloak

4.2.4 Validazione dei manifesti e linting

Lo stage di linting si occupa della verifica statica e sintattica dei file di configurazione Kubernetes, con l'obiettivo di intercettare errori prima che possano compromettere il deploy.

In particolare, vengono analizzati:

- Il chart Helm del progetto
- I file YAML relativi a componenti come Cert-Manager e Vault-CSI

Il job `lint_helm_and_manifests` impiega due strumenti principali:

- *helm lint*, per verificare la correttezza sintattica e la coerenza dei template all'interno del chart Helm;
- *yamllint*, per effettuare controlli formali sulla struttura dei file YAML, come l'uso corretto degli spazi, la coerenza nei rientri e l'integrità della sintassi.

In caso di problemi, la pipeline viene interrotta, impedendo che manifesti mal formati vengano applicati al cluster Kubernetes.

```
lint_helm_and_manifests:
  stage: lint
  image: alpine:latest
  before_script:
    - apk add --no-cache bash curl git tar python3 py3-pip openssl
    - pip install --break-system-packages yamllint
    - curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
  script:
    - helm lint kubernetes/chart
    - yamllint kubernetes/certificate/
    - yamllint kubernetes/vault-csi/
  needs:
    - job: docker_push_keycloak
      artifacts: false
    - job: docker_push_express
      artifacts: false
  only:
    - push
```

Figura 4.7: Job `lint_helm_and_manifests`

4.2.5 Deploy

Lo stage di deploy ha il compito di applicare in modo completamente automatizzato i manifest al cluster Kubernetes. L'intera operazione viene gestita tramite Helm e si integra con l'API di *DigitalOcean* per ottenere dinamicamente il *kubeconfig* necessario alla connessione con il cluster.

Le attività principali del job *helm_deploy_backend* includono:

- Il download del kubeconfig tramite un'API autenticata di DigitalOcean, che consente alla pipeline di connettersi in modo sicuro al cluster di destinazione;
- L'aggiornamento dinamico dei valori relativi all'immagine e alla versione nel file *values.yaml* del chart Helm, utilizzando i riferimenti delle immagini appena buildate e taggate;
- L'esecuzione del comando *helm upgrade --install* per installare o aggiornare l'applicazione;
- L'applicazione dei file YAML associati a Cert-Manager e Vault CSI tramite *kubectl apply*.

L'approccio seguito si ispira al modello GitOps, dove la configurazione viene tenuta sotto controllo come se fosse codice, rendendo tutto più trasparente e facile da gestire.

```

helm_deploy_backend:
  stage: deploy
  image: alpine:latest
  services:
    - docker:dind
  variables:
    DOCKER_HOST: tcp://docker:2375/
    DOCKER_TLS_CERTDIR: ""
  before_script:
    - apk add --no-cache curl bash git openssl jq
    - curl -L https://github.com/mikefarah/yq/releases/latest/download/yq_linux_amd64 -o /usr/local/bin/yq
    - chmod +x /usr/local/bin/yq
    - curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
    - curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl" | bash
    - chmod +x kubectl && mv kubectl /usr/local/bin/
    - |
      CLUSTER_ID=$(curl -s -X GET \
        -H "Authorization: Bearer $DO_API_TOKEN" \
        "https://api.digitalocean.com/v2/kubernetes/clusters" \
        | jq -r '.kubernetes_clusters[] | select(.name == env.DO_CLUSTER_NAME) | .id')

    curl -s -X GET \
      -H "Authorization: Bearer $DO_API_TOKEN" \
      "https://api.digitalocean.com/v2/kubernetes/clusters/$CLUSTER_ID/kubeconfig" \
      -o kubeconfig

    export KUBECONFIG=$CI_PROJECT_DIR/kubeconfig
  script:
    - export KUBECONFIG=./kubeconfig
    - yq e '.express.image = "'$CI_REGISTRY_IMAGE/express'" -i kubernetes/chart/values.yaml
    - yq e '.express.version = "'$CI_COMMIT_SHORT_SHA'" -i kubernetes/chart/values.yaml
    - yq e '.keycloak.image = "'$CI_REGISTRY_IMAGE/keycloak'" -i kubernetes/chart/values.yaml
    - yq e '.keycloak.version = "'$CI_COMMIT_SHORT_SHA'" -i kubernetes/chart/values.yaml
    - helm upgrade --install app kubernetes/chart -n app -f kubernetes/chart/values.yaml
    - kubectl apply -f kubernetes/certificate/
    - kubectl apply -f kubernetes/vault-csi/
  needs:
    - job: lint_helm_and_manifests
      artifacts: false
  only:
    - push

```

Figura 4.8: Job helm_deploy_backend

4.2.6 Kubernetes Security

Dopo il deploy, la pipeline esegue una scansione della sicurezza del cluster Kubernetes tramite Kubescape, uno strumento open-source sviluppato da ARMO per il *Kubernetes Security Posture Management* (KSPM). Kubescape analizza lo stato effettivo del cluster e le configurazioni applicate, confrontandole con framework di sicurezza riconosciuti come NSA-CISA Kubernetes Hardening Guidance, MITRE

ATT&CK, CIS Kubernetes Benchmark e NSA-Hardening Framework. Lo strumento è in grado di rilevare problematiche comuni come container eseguiti con privilegi elevati, l'assenza di SecurityContext, ingress privi di protezione TLS, ruoli eccessivamente permissivi e policy di rete troppo aperte.

Nella pipeline, Kubescape viene eseguito con il profilo NSA, che effettua verifiche su vari aspetti di sicurezza, tra cui:

- configurazione dei permessi RBAC,
- utilizzo del readOnlyRootFilesystem,
- presenza di runAsNonRoot,
- limitazione delle capability nei container,
- conformità delle risorse ai requisiti minimi di hardening.

I risultati della scansione vengono esportati in formato HTML per essere utilizzato nei report assemblati o per un consulto manuale, e quindi archiviati come artifact della pipeline.

```

kubescape_scan:
  stage: kube-security
  image: alpine:latest
  services:
    - docker:dind
  variables:
    DOCKER_HOST: tcp://docker:2375/
    DOCKER_TLS_CERTDIR: ""
  before_script:
    - apk add --no-cache curl bash git openssl jq
    - curl -s https://raw.githubusercontent.com/kubescape/kubescape/master/install.sh | /bin/bash
    - |
      CLUSTER_ID=$(curl -s -X GET \
        -H "Authorization: Bearer $DO_API_TOKEN" \
        "https://api.digitalocean.com/v2/kubernetes/clusters" \
        | jq -r '.kubernetes_clusters[] | select(.name == env.DO_CLUSTER_NAME) | .id')

      curl -s -X GET \
        -H "Authorization: Bearer $DO_API_TOKEN" \
        "https://api.digitalocean.com/v2/kubernetes/clusters/$CLUSTER_ID/kubeconfig" \
        -o kubeconfig

      export KUBECONFIG=$CI_PROJECT_DIR/kubeconfig
  script:
    - export KUBECONFIG=./kubeconfig
    - kubescape scan framework nsa --format json --format-version v2 --output results.json
    - kubescape scan framework nsa --format html --output kubescape_report.html
    - |
      CRITICAL=$(jq '[.summaryDetails.resourcesSeverityCounters.criticalSeverity] | add' results.json)
      HIGH=$(jq '[.summaryDetails.resourcesSeverityCounters.highSeverity] | add' results.json)

      echo "Critical: $CRITICAL | High: $HIGH"
      if [ "$CRITICAL" -gt 0 ] || [ "$HIGH" -gt 0 ]; then
        echo "High/Critical Vulnerabilities Found."
        exit 1
      else
        echo "Vulnerabilities Not Found."
      fi
  artifacts:
    when: always
    paths:
      - kubescape_report.html
  needs:
    - job: helm_deploy_backend
  only:
    - push

```

Figura 4.9: Job kubescape_scan

4.2.7 DAST – Dynamic Application Security Testing

Una volta completato il deploy e verificata l'infrastruttura, la pipeline passa all'analisi di sicurezza a runtime, simulando un attacco esterno contro l'applicazione attiva. Questo tipo di controllo, noto come *DAST* (*Dynamic Application Security Testing*), consente di rilevare vulnerabilità che emergono solo durante l'interazione reale con l'applicazione. Tra i problemi più frequenti individuati con questo approccio ci sono:

- attacchi di tipo injection (come SQL o command injection),
- Cross-Site Scripting (XSS),
- autenticazione non sicura,
- redirect non protetti.

Per condurre questo tipo di analisi, viene utilizzato *OWASP ZAP (Zed Attack Proxy)*, uno strumento open-source molto diffuso nel campo della sicurezza delle applicazioni web.

ZAP permette di:

- importare direttamente le specifiche OpenAPI per scansionare in automatico tutte le API esposte,
- simulare l'interazione con l'app attraverso script personalizzati,
- generare report dettagliati delle vulnerabilità trovate, ordinati per livello di gravità.

All'interno della pipeline, ZAP viene eseguito in modalità headless, seguendo le istruzioni definite in un file *zap_config.yaml*. Questo file descrive il contesto di analisi, specifica le rotte da includere o escludere, e configura il metodo di gestione della sessione. Per l'autenticazione viene usato uno script JavaScript personalizzato (*auth_script.js*), eseguito tramite l'engine Graal.js: lo script effettua il login, estrae il cookie accessToken dalla risposta, e lo applica alle richieste successive.

L'avvenuto login viene poi verificato tramite una chiamata a un endpoint specifico.

Per facilitare e rendere più completa la scansione, viene anche caricato un file `openapi.json`, che descrive in dettaglio le API REST disponibili. In questo modo, ZAP può esplorare tutti gli endpoint senza bisogno di analisi manuali, aumentando la copertura del test.

Infine, viene avviata una scansione attiva (*activeScan*) utilizzando l'utente autenticato e le policy predefinite. Al termine, ZAP produce un report HTML che viene salvato tra gli artifact della pipeline.

```

dast_scan:
  stage: dast_zap
  image: gher.io/zaproxy/zaproxy:stable
  allow_failure: false
  script:
    - mkdir -p /zap/wrk
    - cp zap_config.yaml /zap/wrk/
    - cp auth_script.js /zap/wrk/
    - cp openapi.json /zap/wrk/
    - zap.sh -daemon -port 8090 -host 0.0.0.0 -config api.disablekey=true -config api.addrs.addr.name="*" -config api.addrs.addr.regex=true -addoninstall authhelper -addoninstall graaljs &
    - sleep 30
    - curl "http://localhost:8090/JSON/openapi/action/importFile/?file=/zap/wrk/openapi.json"
    - sleep 10
    - curl -X POST "http://localhost:8090/JSON/automation/action/runPlan/" -d "filePath=/zap/wrk/zap_config.yaml"
    - |
      while true; do
        finished=$(curl -s "http://localhost:8090/JSON/automation/view/planProgress/?planId=0" | jq -r '.finished')
        if [ -n "$finished" ]; then
          echo "Scan completed at: $finished"
          break
        fi
        sleep 2
      done
    - curl "http://localhost:8090/OTHER/core/other/htmlreport/" -o zap_report.html
  artifacts:
    when: always
    paths:
      - zap_report.html
  needs:
    - job: kubescape_scan
  only:
    - push

```

Figura 4.10: Job `dast_scan`

4.3 Raccolta dei Report

A chiusura del processo, la pipeline esegue uno stage conclusivo chiamato *collect_reports*, pensato per raccogliere tutti i risultati generati durante le fasi di analisi precedenti e riunirli in un report unico e con-

sultabile. Il job è configurato per essere eseguito sempre, indipendentemente dall'esito degli altri stage. Anche in presenza di vulnerabilità gravi o errori, il sistema garantisce la generazione di un resoconto completo.

Il task viene svolto all'interno di un container Python che include la libreria *pandas*, utilizzata per leggere i report in formato JSON e HTML e fonderli in un singolo file chiamato *combined_report.html*. Il documento generato funge da dashboard interattiva, organizzando i risultati in tabelle e visualizzazioni strutturate.

Tra i report analizzati troviamo:

- *ESLint* e *NodeJsScan* per l'analisi statica del codice JavaScript,
- *Trivy*, utilizzato per la scansione delle immagini Docker di Express e Keycloak,
- *Kubescape*, che valuta lo stato di sicurezza del cluster Kubernetes,
- *OWASP ZAP*, per il controllo dinamico dell'applicazione già in esecuzione.

I file JSON vengono elaborati per estrarre i dati rilevanti e mostrarli in tabelle HTML, con evidenziazione automatica delle vulnerabilità in base alla severità. I report in HTML, come quelli prodotti da ZAP o Kubescape, vengono invece inclusi tramite *iframe*, consentendo di

sfogliarli direttamente all'interno del file aggregato, senza doverli aprire separatamente.

4.4 Esecuzione e Mitigazione

Una volta definita l'intera pipeline CI/CD secondo i principi del *SecDevOps*, viene eseguita al completo per verificare che ogni fase – dalla scansione statica fino ai test dinamici a runtime – sia correttamente integrata e funzionante. Durante l'esecuzione iniziale, alcuni stage della pipeline si interrompono in corrispondenza della rilevazione di vulnerabilità o configurazioni non conformi. Questo rappresenta uno dei punti di forza dell'approccio SecDevOps: intercettare tempestivamente le criticità prima che raggiungano ambienti di produzione.

In questa sezione, vengono analizzati in dettaglio i punti di blocco individuati durante il primo run della pipeline. Per ciascuno di essi si illustrano:

- Le cause del fallimento o delle segnalazioni critiche,
- I contenuti generati nei report corrispondenti,
- Le azioni di mitigazione adottate (modifiche al codice, aggiornamenti di immagini, rafforzamento della configurazione),
- E infine, il completamento della pipeline con successo dopo l'applicazione delle correzioni.

4.4.1 Fase SAST – Errori rilevati da ESLint

Durante la prima esecuzione della pipeline, uno dei primi ostacoli si è verificato nello stage di analisi statica del codice (*SAST*), a causa di diversi errori segnalati da ESLint. Lo strumento ha individuato numerose problematiche all'interno del codice JavaScript dell'applicazione Express, tra cui l'uso di variabili non definite, dichiarazioni inutilizzate, e incoerenze stilistiche legate a spaziature o convenzioni di scrittura. Un esempio ricorrente riguardava l'utilizzo di *var* al posto di *let* o *const*, pratica sconsigliata perché meno sicura e meno chiara.

Per risolvere queste criticità, il primo passo è stato quello di installare ESLint come dipendenza di sviluppo (*-save-dev*) all'interno del progetto. Si è quindi proceduto a un'operazione di refactoring del codice, sfruttando il comando *eslint -fix* per correggere automaticamente gran parte degli errori. I problemi rimanenti sono stati risolti manualmente, con attenzione a non compromettere la logica applicativa.

Al termine delle modifiche, l'applicazione è stata testata in locale per assicurarsi che tutto funzionasse correttamente. La pipeline è stata quindi rieseguita, e lo stage di linting è stato superato con successo, confermando che le correzioni apportate erano efficaci e che il codice risultava ora conforme agli standard di qualità previsti.

4.4.2 Fase SAST – Errori rilevati da NodeJsScan

Dopo il superamento dello stage ESLint, la pipeline ha proseguito con l'analisi statica del codice specifica per Node.js, eseguita tramite NodeJsScan. Lo strumento ha analizzato la directory contenente il backend Express, individuando alcune criticità legate alla gestione delle intestazioni HTTP di sicurezza e a un potenziale caso di username hardcoded. Nel dettaglio, uno degli avvisi riguardava una variabile chiamata *username*, presente nel file *auth.js* alla riga 127. Sebbene fosse utilizzata in un contesto di test, la sua denominazione generava un falso positivo, suggerendo la presenza di credenziali statiche nel codice. Per evitare ambiguità, la variabile è stata rinominata in *usernameField*, chiarendo il suo scopo ed eliminando l>alert.

Più rilevante è stata invece la segnalazione dell'assenza di alcune intestazioni HTTP di sicurezza, considerate fondamentali per proteggere le applicazioni web da attacchi comuni come XSS, clickjacking o MIME-sniffing. Tra gli header mancanti segnalati figuravano:

- Content-Security-Policy
- X-Frame-Options
- Strict-Transport-Security
- X-Content-Type-Options
- X-XSS-Protection

- X-Download-Options
- X-Powered-By
- Public-Key-Pins (HPKP)

Per risolvere il problema, è stato aggiunto un middleware personalizzato alla configurazione del server Express, con l'inserimento esplicito degli header necessari:

```
// Security headers middleware
app.use((req, res, next) => {
  res.setHeader('Content-Security-Policy', 'default-src \'none\'');
  res.setHeader('X-Frame-Options', 'DENY');
  res.setHeader('Strict-Transport-Security', 'max-age=63072000; includeSubDomains');
  res.setHeader('X-Content-Type-Options', 'nosniff');
  res.setHeader('X-XSS-Protection', '1; mode=block');
  res.setHeader('X-Download-Options', 'noopen');
  res.removeHeader('X-Powered-By');
  next();
});
```

Grazie a questo intervento, l'applicazione ha acquisito una maggiore resilienza contro diversi vettori d'attacco. L'unico avviso rimasto è relativo all'header *Public-Key-Pins (HPKP)*, che è stato intenzionalmente ignorato poiché deprecato e non più raccomandato, a causa dei rischi operativi che comporta.

Dopo l'introduzione del middleware, la pipeline è stata rieseguita e lo stage *sast_express* ha superato con successo tutte le verifiche, confermando l'efficacia delle correzioni apportate.

4.4.3 Container Scanning – Immagine Keycloak

Durante lo stage di container scanning dedicato all'immagine personalizzata di Keycloak, la pipeline ha individuato una vulnerabilità ad alta gravità, identificata come *CVE-2025-3501*. Il problema interessava la libreria *org.keycloak:keycloak-services* in versione *26.2.0* e riguardava un controllo inadeguato dell'hostname all'interno del modulo *keycloak.protocol.services*. Questo difetto poteva potenzialmente esporre l'applicazione a spoofing o attacchi man-in-the-middle (MitM), in particolare in contesti che prevedono autenticazione federata o deployment distribuiti.

La mitigazione è risultata piuttosto immediata: il pacchetto affetto è stato aggiornato alla versione *26.2.2*, in cui la vulnerabilità era già stata corretta secondo la documentazione ufficiale fornita da Aqua Security. L'intervento è stato effettuato direttamente nel Dockerfile.

4.4.4 Kubernetes Security – Scansione con Kubescape

La verifica della sicurezza dell'infrastruttura Kubernetes è stata affidata a *Kubescape*, lo strumento open source utilizzato per valutare la conformità dei cluster rispetto a benchmark consolidati. Durante l'esecuzione della pipeline, lo scan è stato inizialmente lanciato su tutti i namespace, ma per ridurre il rumore e concentrare l'attenzione sulle sole componenti sviluppate all'interno del progetto, sono stati esclusi i

namespace di sistema e di terze parti riducendo i falsi positivi e focalizzando l'analisi sull'ambiente applicativo effettivamente sotto controllo. Nel namespace app, Kubescape ha individuato due principali criticità:

- **Variabili d'ambiente potenzialmente sensibili:** alcune environment variables sono state segnalate come sospette, in quanto i loro nomi corrispondevano a pattern spesso associati a credenziali o segreti. Pur non contenendo valori realmente sensibili, per maggiore chiarezza e per evitare allarmi futuri, i nomi di queste variabili sono stati rinominati, mantenendo la stessa funzionalità ma migliorando la leggibilità e la compliance.
- **SecurityContext incompleto:** nei deployment di Express e Keycloak mancava il campo *runAsGroup*, consigliato dalle best practice per una corretta separazione dei privilegi. È stato quindi aggiunto con valore *1000*, già coerente con gli altri parametri di sicurezza utilizzati.

A livello di logica di esecuzione, la pipeline è stata configurata in modo da interrompere automaticamente lo stage se anche una sola vulnerabilità classificata come *Critical* o *High* veniva rilevata. Questo ha permesso di forzare l'intervento immediato e di assicurare che nessuna configurazione rischiosa venisse portata in ambienti di produzione.

Per il futuro, sarà possibile estendere l'analisi anche ai namespace esclusi, bilanciando la copertura della sicurezza con l'esigenza di man-

tenere la stabilità del cluster, soprattutto nel caso di eventuali modifiche correttive più invasive.

4.4.5 Dynamic Application Security Testing (DAST)

All'interno della pipeline, una delle ultime fasi è dedicata alla scansione dinamica (DAST) tramite lo strumento *OWASP ZAP*, con l'obiettivo di individuare vulnerabilità durante l'esecuzione reale dell'applicazione, simulando il comportamento di un utente esterno.

La scansione si è conclusa con esito positivo, senza rilevare vulnerabilità classificate come alte o medie. Sono emerse però due segnalazioni di bassa gravità, che sono state valutate ma non hanno richiesto interventi correttivi:

- **Application Error Disclosure:** in alcuni casi l'applicazione ha restituito risposte *HTTP 500* con messaggi di errore potenzialmente utili a un attaccante per dedurre il funzionamento interno del server. Tuttavia, nel contesto attuale e considerando la tipologia di errori, il rischio è stato considerato trascurabile.
- **Cookie con attributo SameSite=None:** sia *accessToken* che *refreshToken* sono stati identificati con l'attributo *SameSite=None*, configurazione necessaria per alcune funzionalità cross-site. Pur essendo evidenziata come potenzialmente rischiosa, è

risultata coerente con le esigenze dell'applicazione e non è stata modificata.

In aggiunta, ZAP ha prodotto alcuni avvisi informativi legati all'autenticazione e alla gestione delle sessioni. Nessuno di questi, però, ha evidenziato vulnerabilità concrete tali da richiedere interventi immediati.

Data la natura a basso impatto delle segnalazioni ricevute e la loro funzione principalmente informativa, non sono state apportate modifiche al codice né alla configurazione dell'infrastruttura in seguito alla scansione DAST.

Capitolo 5

Intrusion Detection e Sicurezza Runtime

Gli *Intrusion Detection System (IDS)* sono strumenti progettati per monitorare sistemi e reti con l'obiettivo di identificare comportamenti anomali che possano indicare tentativi di intrusione o compromissione.

Gli IDS si dividono in:

- *Host-Based IDS (HIDS)*: monitorano i singoli host e i loro eventi di sistema.
- *Network-Based IDS (NIDS)*: analizzano il traffico di rete per rilevare attività sospette.

Approcci: Signature-Based vs Anomaly-Based

- *Signature-Based*: rilevano attacchi noti confrontando i pattern con firme predefinite.

- *Anomaly-Based*: identificano comportamenti anomali rispetto a una baseline appresa.

Nei moderni ambienti cloud-native come Kubernetes, gli IDS tradizionali risultano limitati per mancanza di visibilità all'interno dei container e per l'elevata dinamicità e distribuzione dei workload.

5.1 Tetragon: Sicurezza Runtime con eBPF

Tetragon è un sistema di sicurezza runtime basato su eBPF che monitora eventi del kernel Linux in tempo reale, senza sidecar o agenti nei container. Le attività che può intercettare includono:

- Creazione di processi sospetti (es. shell, comandi remoti),
- Apertura di file sensibili,
- Comunicazioni di rete anomale,
- Escalation di privilegi.

5.1.1 Tecnologia eBPF

Extended Berkeley Packet Filter (eBPF) è una tecnologia del kernel Linux che permette l'esecuzione sicura di codice in risposta a eventi di basso livello, come system call o pacchetti di rete. eBPF è nato

per il networking, ma oggi viene usato per sicurezza, osservabilità e tracciamento nei sistemi Linux. In Kubernetes, consente:

- Tracciamento delle comunicazioni tra pod,
- Monitoraggio dei processi,
- Monitoraggio dei processi,
- Osservazione delle attività filesystem.

Questa tecnologia offre una visibilità profonda senza introdurre overhead, consentendo un monitoraggio efficiente. Non richiede modifiche ai container e garantisce un'esecuzione sicura e isolata all'interno del kernel.

5.1.2 Architettura di Tetragon

Tetragon viene distribuito come *DaemonSet* su tutti i nodi del cluster Kubernetes, agendo come un demone di rete capace di raccogliere e processare gli eventi kernel in tempo reale. La sua configurazione avviene tramite file YAML di tipo `values.yaml`, dove è possibile specificare:

- le policy da applicare,
- i livelli di log desiderati,
- le integrazioni esterne da attivare (es. *Loki*, *Prometheus*),

- le regole di esportazione degli eventi.

Questa configurazione permette un'integrazione nativa con l'infrastruttura di *observability* e sicurezza esistente, senza impattare sulle performance dei container.

```
tetragon:  
  enabled: true  
  enableK8sAPIAccess: true  
  export:  
    # Abilita esportazione in Loki  
    json: true  
    logLevel: info  
    logToStdout: true  
  prometheus:  
    enabled: true  
    serviceMonitor:  
      enabled: true  
      labels:  
        release: kube-prometheus-stack  
  enableProcessCred: true
```

Figura 5.1: Configurazione di Tetragon

La sua architettura può essere schematizzata in tre livelli principali:

- *Tracepoints*: per eventi generici (es. esecuzione processi),
- *Kprobes*: per intercettare chiamate di funzione nel kernel,
- *LSM Hooks*: per controlli di sicurezza (accesso file, privilegi).

Kernel Space

Tetragon inserisce programmi *eBPF* in specifici hook del kernel:

- *tracepoints*: per eventi come `sched_process_exec` (esecuzione di un nuovo processo),

- *kprobes*: per intercettare funzioni del kernel Linux,
- *LSM hooks*: per attività legate alla sicurezza, come accessi a file e privilegi.

Questi hook raccolgono dati su eventi rilevanti come:

- esecuzione di processi,
- accesso a file o socket,
- tentativi di escape dai namespace,
- escalation di privilegi,
- attività di rete (TCP, DNS, HTTP...).

User Space – Tetragon Daemon

Il *Tetragon agent* (un demone che gira come *DaemonSet* nei nodi del cluster) riceve gli eventi raccolti da *eBPF* e:

- li struttura in formato JSON o *Prometheus*,
- li invia a strumenti esterni (*Loki*, *Fluentd*, *SIEM*, *Grafana*),
- applica le policy definite nel cluster per filtrare e reagire agli eventi.

Output e Integrazione

Il sistema esporta:

- log strutturati, utili per analisi forense o alert,

- metriche, integrabili in *Grafana*,
- eventi real-time, utilizzabili in sistemi di notifica (es. *Telegram*).

5.1.3 Politiche di Tracciamento (TracingPolicy)

Una *TracingPolicy* è una risorsa Kubernetes che definisce:

- eventi da intercettare (es. `process_exec`),
- condizioni (es. nome del comando o percorso),
- azioni da compiere (log, blocco, alert).

Esempio: rilevamento shell

- esecuzione di `/bin/sh` o `/bin/bash`,
- policy che logga ed eventualmente allerta.

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: tracepoint-exec
spec:
  tracepoints:
    - subsystem: sched
      event: sched_process_exec
```

Figura 5.2: Esempio

1. Processo eseguito in container,
2. Tracepoint `sched_process_exec` lo intercetta,
3. Programma *eBPF* raccoglie dati,
4. Daemon li elabora,
5. Evento loggato/inviato,
6. Possibile azione (alert, blocco).

5.2 Integrazione con Cilium: Sicurezza e Visibilità di Rete

Cilium è una *Container Network Interface (CNI)* per Kubernetes basata su *eBPF*, che sostituisce `iptables` e `bridge` con programmi *eBPF* per offrire maggiori performance e visibilità.

Offre funzionalità avanzate di sicurezza:

- *NetworkPolicy* L3/L4/L7,
- ispezione *HTTP*, *DNS*, *Kafka*,
- regole *DNS-aware*,
- integrazione con *Hubble* e *Grafana*.

Nel progetto, *Cilium* è la CNI predefinita del cluster *DigitalOcean*, rendendo l'integrazione con *Tetragon* immediata.

I benefici dell'integrazione includono:

- visibilità completa (rete + kernel),
- correlazione tra eventi di rete e comportamenti dei processi,
- nessun bisogno di *sidecar* o polling,
- osservabilità proattiva e reattiva.

Caso pratico:

- Cilium applica una policy che impedisce al pod A di accedere al pod B sulla porta TCP 5432,
- un attaccante compromette il pod A, apre una shell e lancia un port scan,
- Tetragon rileva l'apertura della shell e la creazione del socket,
- l'evento viene loggato e notificato, senza overhead.

5.3 Monitoraggio degli Accessi Runtime

Il progetto è stato realizzato su un cluster *Kubernetes* ospitato su *DigitalOcean*, dove sono stati deployati diversi pod per comporre un'infrastruttura di microservizi tipica. Tra i principali componenti del cluster:

- *NGINX* come *Ingress Controller*,

- *Node.js* (*Express*) come applicazione web esposta pubblicamente,
- *Keycloak* per l'autenticazione e gestione delle identità,
- *PostgreSQL* come database persistente.

Il punto di ingresso principale del cluster è rappresentato dal servizio *Express*. Proprio da questo punto può originarsi un attacco, specialmente in caso di vulnerabilità come la *Remote Code Execution (RCE)*. In scenari del genere, l'uso non autorizzato di comandi `shell` o l'abuso del comando `kubectl exec` potrebbero compromettere l'intero cluster.

5.3.1 Obiettivo del Monitoraggio

Per prevenire e rilevare tempestivamente questi attacchi, è stato progettato un sistema in grado di:

- monitorare in tempo reale l'esecuzione di comandi sospetti (es. `/bin/sh`, `bash`) nel pod *Express*,
- inviare una notifica istantanea via *Telegram* all'amministratore del cluster,
- salvare log strutturati e persistenti su *Loki* per l'analisi successiva e la correlazione con altri eventi.

5.3.2 Implementazione della Soluzione

La rilevazione degli eventi si basa su una *TracingPolicy* di *Tetragon*, che intercetta specifiche esecuzioni di processo. Una query su *Loki* consente di estrarre rapidamente questi eventi:

```
{namespace="tetragon"} |= "process_exec" |~  
"/bin/(sh|bash)" |~ "express"
```

Uno script *Python* è stato sviluppato per:

- interrogare periodicamente *Loki* tramite la query *LogQL*,
- identificare nuovi eventi corrispondenti ai pattern sospetti,
- inviare una notifica *Telegram* con i dettagli dell'evento.

Lo script è stato containerizzato e schedulato nel cluster come *Kubernetes Job*, in modo da renderlo portabile e automatizzarne completamente l'esecuzione.

5.3.3 Risultati Ottenuti

Questa integrazione ha prodotto una soluzione semplice, efficace e altamente reattiva:

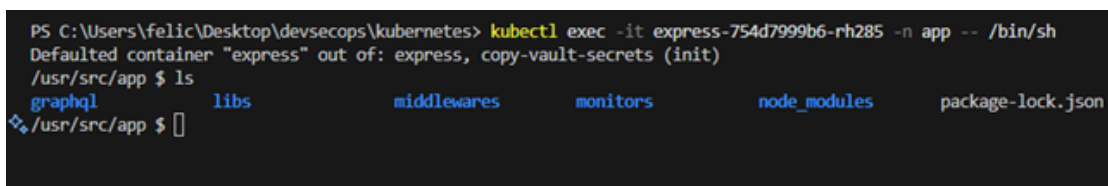
- qualsiasi apertura di shell nel pod *Express* viene rilevata entro pochi secondi,

- gli eventi vengono tracciati in *Loki* e restano disponibili per operazioni di *forensics* e *auditing*,
- l'amministratore riceve notifiche su *Telegram* in tempo reale, con dettagli utili all'identificazione immediata del rischio.

Come si può osservare dalle immagini seguenti, è stata eseguita una shell sul pod *Express* tramite il comando. Dopo l'apertura della shell, è stato eseguito anche un comando `ls`. *Tetragon* ha intercettato entrambi gli eventi in tempo reale.

Di seguito, l'output notificato dallo script *Python* via *Telegram*, che mostra i dettagli completi dell'esecuzione sospetta. Questi screenshot dimostrano l'efficacia del monitoraggio automatico e la precisione dell'integrazione tra *Tetragon*, *Loki* e *Telegram* nella rilevazione di attività potenzialmente malevole nel cluster *Kubernetes*.

Questa soluzione rappresenta un valido esempio di rilevamento proattivo e risposta automatizzata in un cluster *Kubernetes* moderno, integrando strumenti *open-source* in modo coerente e leggero.



```
PS C:\Users\felic\Desktop\devsecops\kubernetes> kubectl exec -it express-754d7999b6-rh285 -n app -- /bin/sh
Defaulted container "express" out of: express, copy-vault-secrets (init)
/usr/src/app $ ls
graphql      libs          middlewares   monitors      node_modules  package-lock.json
❖ /usr/src/app $
```

Figura 5.3: Esecuzione di un comando su Express

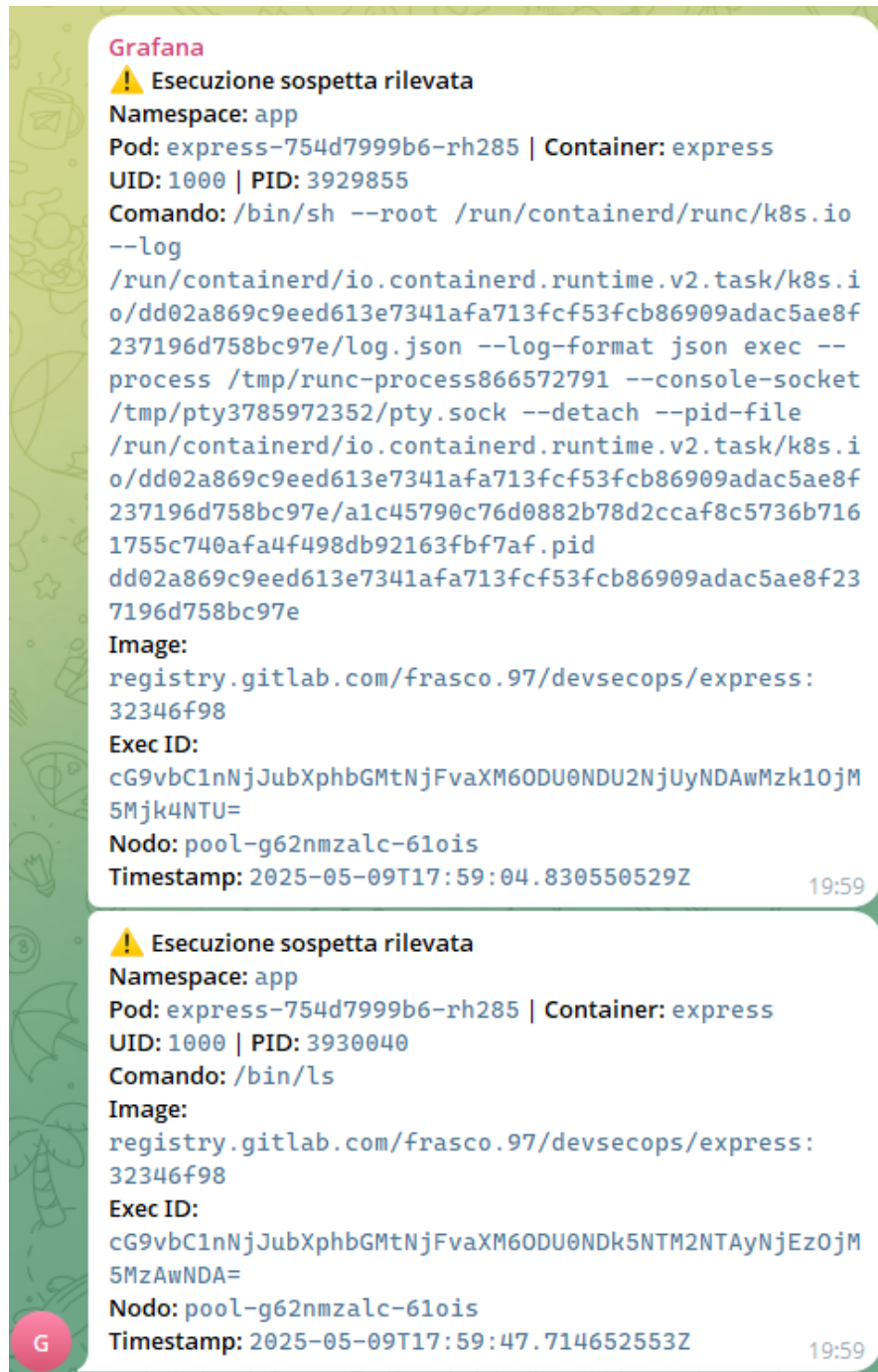


Figura 5.4: Alert dell'azione su Express in Telegram