UNIVERSITÀ DEGLI STUDI DI CATANIA

Dipartimento di Matematica e Informatica

Corso di Laurea Triennale in Informatica

Giuseppe Bonanno

PACKS (Packaging Automated for Kuberneters Containers System)

Relazione Progetto Finale

Relatore: Prof. Nicotra Salvatore

Anno Accademico 2023 - 2024

Prima di proseguire con la trattazione, desidero dedicare alcune parole alle persone senza le quali non sarei riuscito ad arrivare fin qui.

In primo luogo, un sentito ringraziamento va al mio relatore, il Prof. Salvatore Nicotra. Grazie alle sue lezioni, ho scoperto la passione per nuove tecnologie, che mi hanno stimolato ad approfondire temi affascinanti e a metterli in pratica in questo progetto. Il suo supporto e la sua guida sono stati fondamentali per la realizzazione di questo lavoro.

Un grazie di cuore va ai miei genitori, che con il loro sostegno costante mi hanno permesso di raggiungere questo traguardo, superando insieme tutti gli alti e bassi che hanno costellato questo percorso.

Ai miei fratelli, a cui voglio un mondo di bene, auguro con tutto me stesso di trovare la loro strada, qualunque essa sia. Che sia un cammino capace di guidarli verso una vita piena di gioie, conquiste e soddisfazioni.

Infine, desidero abbracciare idealmente tutte le persone, vicine e lontane, che mi hanno supportato e che hanno creduto in me. La vostra presenza e il vostro affetto sono stati una forza inestimabile, a voi va tutta la mia più profonda gratitudine.

Abstract

Questo progetto propone di risolvere le limitazioni hardware che diversi studenti universitari incontrano nello sviluppo e soprattutto nel testing di applicazioni containerizzate. Tale sistema, denominato "PACKS", offre una piattaforma web che permette di caricare, eseguire e visualizzare i propri container in un ambiente Kubernetes centralizzato e controllato, sfruttando le risorse hardware significativamente superiori a quelle che, in media, sono disponibili sui laptop personali. Attraverso l'uso di Helm, il sistema astrae la complessità di Kubernetes, permettendo agli utenti di utilizzare una sintassi semplificata, ispirata a Docker Compose, per definire le proprie applicazioni. La piattaforma web offre un interfaccia semplice ed intuitiva che consente di caricare, avviare, visualizzare log e collegarsi alle porte dei container. Inoltre gli studenti possono lavorare in uno spazio personale con la possibilità di consegnare i loro progetti al docente per il testing e la valutazione finale. L'approccio adottato mira a semplificare ed eliminare numerosi passaggi normalmente richiesti agli studenti durante lo sviluppo, con l'obiettivo principale di eliminare completamente la necessità di disporre di un computer con elevate capacità hardware.

Indice

Capitolo 1

Introduzione

Negli ultimi anni, l'uso dei container nel mondo informatico ha cambiato radicalmente quello che è stato, e che tutt'oggi è, il mondo dei sistemi distribuiti. Sin dai primi linguaggi di programmazione, l'astrazione è stata fondamentale nella loro evoluzione. Si è passati dalla classica programmazione batch a quella Object Oriented, fino ad arrivare ad astrarre l'intera architettura attraverso l'introduzione della virtualizzazione e successivamente attraverso l'uso dei container. Questa evoluzione ha portato alla creazione di ambienti e strutture sempre piu complesse che oggi formano l'astrazione per eccellenza: il Cloud.
Tali innovazioni hanno portato migliorie in ogni ambito, dalla spiccata sicurezza informatica sino al grande efficientamento prestazionale ed economico, permettendo alle aziende ma anche alle piccole realtà personali di "affidare" ad un astrazione l'esecuzioni del proprio codice senza minimamente considerare la base su cui girerà, la manutenzione che questa richiede e senza gestire scalabilità, fault tolerance, networking e sicurezza ad attacchi informatici. Cosi facendo i programmatori possono liberamente concentrarsi nel produrre solo codice di qualità.
L'altro lato della medaglia invece mostra uno spiccato aumento in verticale della curva di apprendimento per chi si interfaccia verso questo mondo. Quindi oltre a semplificare sempre di più con le astrazioni che nascondono una struttura interna sempre più intricata e complessa, bisogna anche "ammorbidire" questa curva facilitando l'apprendimento sin dalle basi. Quindi vi è un importante esigenza in una soluzione che semplifichi l'approccio alla struttura nascosta e che aiuti nello studio di come questa sia in grado di offrire certi vantaggi.
Ogni applicativo che deve essere seguito ha bisogno di una infrastruttura sottostante, un server, che richiede a sua volta uan gestione e configurazione specifica. Invece l'elaborazione "senza server", o meglio conosciuta come

"serverless computing", è un modello di sviluppo del software dove l'infrastruttura su cui questo è in esecuzione è puramente gestita da terze parti. In questo testo verrà spiegato meglio cosa si intende e verrà fatta una panoramica più ampia di quelli che sono i principali ambienti serverless e chi sono i provider maggiori di tali "servizi".
In passato il software, una volta pronto, veniva rilasciato e distribuito in un pacchetto che doveva essere installato sulla macchina, o server, che a sua volta necessitava di configurazioni e applicativi specifici per il funzionamento. Queste operazioni di configurazione venivano ripetuti ad ogni installazione per ogni macchina. L'avvento dei container portò un grosso vantaggio da questo punto di vista, annullando la ripetitività dell'operazione, dei potenziali svantaggi che questo porta e garantendo un ambiente isolato, leggero e facilmente avviabile in ogni macchina. Fu cosi che si passò dal concetto di server fisico a quello di "server logico".
Più applicativi e componenti che necessitano di comunicare costantemente fra loro possono essere isolati e avviati in diversi container, quindi diversi server logici. Per fare ciò, negli anni, sono nati diversi sistemi di gestione container che ne curano il deployment collettivo, la comunicazione, gli errori, la scalabilità e la suddivisione di risorse. Il più famoso di questi è Kubernetes, una piattaforma open-source per la gestione di servizi e applicativi containerizzati, estensibile e facilmente scalabile. Tutt'oggi i principali servizi ed applicativi più famosi girano su ambienti completamente virtualizzati, in cui più server logici sono eseguiti da una CPU e gestiti da un software come Kubernetes. Il salto di difficoltà che comporta la creazione di un sistema di container che collaborano e contribuiscono ad un applicativo finale è grande tanto quanto l'importanza dei vantaggi che questo sistema offre.

Il progetto si pone come obiettivo principale quello di permettere ad utenti e/o studenti di fare "deploying", attraverso un linguaggio dichiarativo abbastanza intuitivo, del proprio applicativo, formato da più server logici in un ambiente serverless. Far girare il deployment di container in un ambiente centralizzato e non in un ambiente "on-premise", come potrebbe essere il laptop dello studente, porterebbe a risolvere importanti problemi di limitazioni hardware. PACKS, raggruppa e semplifica, attraverso un'interfaccia web, tutti i macchinosi comandi che l'utente compie ripetutamente per connettersi a porte preimpostate, visualizzare log, avviare, eliminare e fermare i vari container.
Kubernetes è lo scheletro di tutto, in esso girano sia i container delle componenti PACKS, sia quelli degli utenti, ovviamente in ambienti logici separati. Kubernetes stesso utilizza un linguaggio dichiarativo formato da diversi tem-

plate e componenti che per un neofita potrebbero essere piuttosto complicati da comprendere, da creare e da utilizzare. Qui entra in gioco Helm, un gestore di pacchetti per Kubernetes, che permette di impacchettare una serie di componenti in un unico grande template. Grazie ad esso, PACKS usa un template Helm predefinito, creato appositamente per gestire componenti e container singoli astraendo completamente la complessità sottostante grazie ad un linguaggio ed una sintassi che ricorda il celebre Docker Compose. Utilizzando questa sintassi e questo template "all in one", gli studenti novizi non sono più obbligati ad affrontare tale scoglio implementativo e possono usare un linguaggio più congeniale per chi è alle prime armi. In questo testo verranno analizzati e discussi più nel dettaglio i container, gli ambienti serverless, Kubernetes e Helm. Successivamente verrà illustrata la struttura di PACKS e le componenti interne, ponendo più attenzione sulle funzionalità e sull'implementazione.

Capitolo 2

Container ed Ambienti Serverless

In questo capitolo verranno approfonditi i concetti di "container" e di "Serverless".

2.1 Container

2.1.1 Definizione e vantaggi

Secondo la definizione fornita da Google stessa, "I container sono pacchetti leggeri del codice dell'applicazione e di dipendenze come versioni specifiche di runtime e librerie dei linguaggi di programmazione necessari per eseguire i servizi software." [1] Infatti, essi sono dei veri e propri server in formato logico perchè impacchettano intorno al software anche le librerie di sistema che diverse macchine non potrebbero aver installate di default, i quali, quindi, richiederebbero un adattamento per ognuna di esse in base alle dipendenze già presenti e mancanti. Tutto parte dall'applicazione, dal file binario che avviamo e che ricerca, oltre a tali librerie, variabili d'ambiente ed altri file nel sistema, essenziali al suo funzionamento. Magari nella macchina usata dallo sviluppatore vi sono presenti e si trovano nei percorsi corretti con le versioni corrette. Ma se dovessimo avviare tale file binario in un altro ambiente vi è la possibilità che qualche libreria, file e/o dipendenza manchi, comportando il crash dell'applicativo o addiritura il non avvio.
Vi è addirittura la possibilità poco remota che vi siano le giuste dipendenze e che l'applicativo non funzioni correttamente perchè le versioni volute da quest'ultimo e quelle presenti, differiscono. Questo è uno dei motivi per il quale conviene impacchettare, oltre ai file dell'applicativo, anche il sistema

che ha attorno: librerie, file e variabili corrette e nella giusta versione. Come si vedrà in seguito, questa strategia permette una maggiore granularità del sistema ed una significativa facilità nell'installazione e nell'avvio di tali ambienti su macchine ben diverse.
Un altro grosso pregio della "containerizzazione" è il poter offrire un servizio e poterlo esporre all'esterno senza preoccupazioni dato che esso gira in un ambiente isolato dal server sottostante ed invisibile all'utilizzatore, offrendo una maggiore sicurezza da attacchi verso la macchina "ospitante".

2.1.2 Differenza fra Macchina Virtuale e Container

Dalla descrizione precedente sembra di star descrivendo una macchina virtuale con un applicativo al suo interno. In effetti, entrambe offrono un ambiente isolato e che comprende tutte le dipendenze e le configurazioni del sistema ed entrambi offrono granularità e sicurezza. Ma la differenza sostanziale sta "al di sotto".
La Virtual Machine simula in tutto e per tutto un server fisico con una propria architettura. Quindi, se vengono avviate piu VM, ognuna di esse avrà il proprio sistema operativo che gira sul proprio kernel che, a sua volta, ha le proprie istruzioni macchina in base all'architettura sottostante. Quest'ultime in particolare vengono gestite e tradotte dall Hypervisor in linguaggio macchina del calcolatore ospitante. Alla creazione, l' Hypervisor stesso assegna ad ogni Macchina Virtuale quantità statiche di risorse hardware. Quindi questo comporta poca flessibilità nella gestione della CPU e della memoria centrale utilizzata. Inoltre, se l'Hypervisor è di tipo 2, le istruzioni ricevute dalle Macchine Virtuali saranno passate al Sistema Operativo sottostante introducendo un altro potenziale collo di bottiglia. Invece i container "condividono" il kernel con il sistema operativo ospitante quindi sono estremamente più leggeri. I processi, le risorse e le interfacce di rete da loro usate vengono isolate grazie a strumenti offerti dal sistema, come Namespace e Cgroups (o Control Groups). Non girando su un Hypervisor i container hanno l'unica limitazione che li contraddistingue dalle Macchine Virtuali, ossia l'obbligo di poter funzionare solo su macchine con egual architettura. Ma nonostante questa limitazione non sia da poco, in realtà, non ha mai generato problemi e, tutt'oggi, il loro uso è largamente diffuso nel mondo dell'Information Technology.

2.1.3 Docker

Docker è la piattaforma software open source per eccellenza che permette ad uno sviluppatore la creazione, l'avvio e la gestione di applicazioni e ser-

vizi containerizzati. Le immagini ed i layer di Docker sono il formato di containerizzazione più diffuso in tutto il mondo tanto che è diventato un vero e proprio standard per l'OCI (Open Container Initiative) e sono usati in qualsiasi contesto che utilizzi container (anche Kubernetes). Innanzitutto distinguiamo e descriviamo cosa sono in particolare i layer e le immagini.

2.1.3.1 Immagini e Layer

Un immagine Docker è una rappresentazione statica e dichiarativa di un sistema operativo e di un file system che contiene tutto il necessario per eseguire l'applicazione correttamente. Quindi, come descritto in precedenza, si tratta di un sistema con tutte le dipendenze corrette e necessarie. Una volta creata un immagine, essa è immutabile, ovvero, non è altro che un insieme di "layer", quindi di strati, che la compongono. Se si vuol modificare un layer sottostante bisogna partire dall'immagine, aggiungerne ulteriori e sovrascriverne altri. Un layer non è altro che una modifica o un aggiunta al file system, quindi, un'installazione di una dipendenza o l'assegnazione di una variabile d'ambiente ne formano uno. Questi sono descritti in un Dockerfile come nel codice 2.1,

Codice 2.1: esempio Dockerfile

```
1 FROM node:latest
2 COPY . /kluster-ui
3 WORKDIR /kluster-ui
4 RUN npm install
5 RUN npm run build
6 EXPOSE 3000
7 ENTRYPOINT ['/bin/bash', '-c', 'npm run start']
```

Prendendo come esempio il Dockerfile mostrato nel codice 2.1, è possibile notare che ogni riga corrisponde a un comando, e ciascuno di essi genera un nuovo layer nell'immagine finale. Questo specifico Dockerfile è utilizzato nel progetto PACKS per costruire l'immagine container della web app. In particolare, il processo inizia con la direttiva "FROM", che specifica come base un'immagine di NodeJS nella sua versione più recente. Successivamente, vengono aggiunti nuovi strati: si copiano i file sorgente, si eseguono le installazioni delle dipendenze con la direttiva "RUN ¡command¿", si espone la porta 3000 con "EXPOSE", e infine si definisce "l'ENTRYPOINT", il comando predefinito che viene eseguito all'avvio del container. Quest'ultima istruzione rappresenta il "punto d'ingresso" del container, indicando quale processo verrà avviato quando il container è eseguito.
Per generare il binario dell'immagine appena definita basta avviare

```
1 docker build /path/to/image/folder -t <tag-immagine>
```

Il container è un'istanza di esecuzione di un'immagine, infatti se ne possono eseguire diversi che si basano su quest'ultima, poichè è mutabile nella sua esecuzione. Oltre ad offrire tutte queste proprietà, il sistema a layer è utile anche per il caching Docker nel 'building', ovvero, nella costruzione. Questo infatti memorizza in locale tutti i layer in modo tale che, se vi è una modifica, non vi sia bisogno di ricostruire il tutto con i layer precedenti, ma può saltare direttamente a quello nuovo. L'immutabilità consente un sistema di versioning delle immagini, infatti vi sono numerosi registry che ne implementano questa particolarità. Il piu usato fra questi è DockerHub, dove è possibile trovare immagini ufficiali, già pronte, di diversi software e sistemi operativi.

2.2 Serverless e Cloud

Il termine Serverless non bisogna intenderlo in modo letterale come 'assenza di server'. In realtà sta ad indicare l'assenza di tutti quei processi piuttosto complicati di mantenimento e gestione di una struttura, la quale deve essere sempre funzionante, efficiente e disponibile. Ma per parlare in modo dettagliato degli ambienti Serverless bisogna prima fare un passo indietro introducendo il Cloud, quello che offre e come ha cambiato del tutto il mondo dell'Information Technology.

2.2.1 Cloud

Fornire una definizione precisa e universale del Cloud è complesso, ma si potrebbe affermare che oggi qualsiasi servizio, elaborazione dati o richiesta, che può essere eseguita localmente, è altrettanto realizzabile in remoto grazie ai servizi Cloud. In altre parole, una delle principali caratteristiche è la possibilità di eseguire operazioni senza la necessità di disporre di hardware o macchine in locale, affidandosi completamente a risorse e infrastrutture fornite in remoto. Innanzitutto, data la vastità dei servizi offerti che è possibile trovare oggi, bisogna distinguerne le principali categorie e livelli, soffermandosi soprattutto sul concetto di Alta Affidabilità, proprietà alla base di tutto.

2.2.2 Alta Affidabilità

L'alta affidabilità (o High Availability) è una proprietà che garantisce "fault tolerance", ossia resistenza e prontezza di soluzione alle interruzioni o errori che possono capitare quotidianamente nei sistemi informatici per i motivi più disparati, come crash del sistema o dell'applicazione, guasti elettrici di

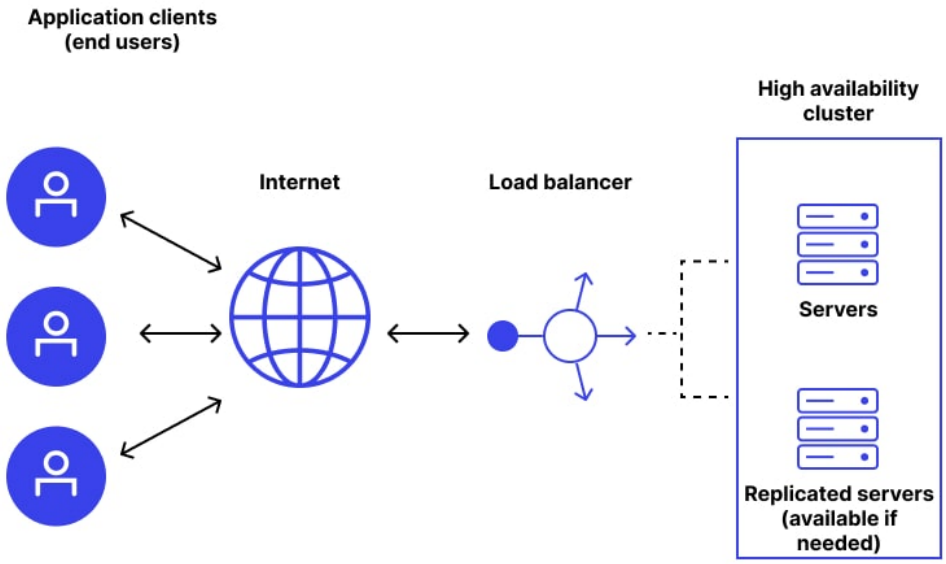

Figura 2.1: Esempio di ridondanza semplice, fonte: [2]

qualsiasi genere e così via. Nel dettaglio questa macroproprietà è formata a sua volta da delle microproprietà, vale a dire: ridondanza, failover, load balancing, monitoring e disaster recovery.

2.2.2.1 Ridondanza

La ridondanza è uno dei concetti chiave del HA. Attraverso la duplicazione delle componenti informatiche come server, loadbalancer e soprattutto quelle critiche come i database di dati persistenti. Insomma, viene duplicato tutto quello il cui il guasto potrebbe interromperne il servizio. Ovviamente vi sono vari livelli di implementazione di tale proprietà. La duplicazione può partire dagli stessi server fisici e sistemi di alimentazione fino ad arrivare a sistemi virtualizzati, dispositivi di rete e dati.

2.2.2.2 Failover

Il failover funziona esclusivamente se presente anche ridondanza. Infatti questa proprietà garantisce una commutazione dei servizi e delle funzionalità su un sistema secondario ridondante quando quello primario fallisce. Il passaggio avviene in maniera inpercettibile e garantisce tempi di "down" praticamente inesistenti e trascurabili.

2.2.2.3 Load Balancing

Il bilanciamento del carico (o load balancing) del sistema, come si intuisce dal nome, si occupa di bilanciare l'uso delle risorse hardware fra tutte le macchine che svolgono lo stesso applicativo e/o lo stesso servizio. Ciò ovviamente si basa su sistemi ridondanti di tipo AA (o Active-Active) e non di tipo AP (o Active-Passive). Il primo indica un sistema dove la ridondanza è anch'essa attiva ed allegerisce il carico del cluster primario garantendone però la piena capacità di assorbimento in caso di failover. Il secondo, invece, non potrebbe essere utile al bilanciamento dato che il cluster ridondante rimane passivo, attivato soltanto in caso di guasto del primario.

2.2.2.4 Monitoring

Fanno parte del monitoring i sistemi che si occupano di "osservare", o appunto monitorare, i sistemi che svolgono un funzione specifica e, di conseguenza, intraprendere l'azione necessaria al caso, specialmente se viene intaccata la funzionalità e la prontezza del sistema. Se ad esempio un server di monitoring vede che un nodo ha pochissima RAM disponibile o un alto utilizzo della CPU, potrebbe decidere di scalare orizzontalmente con altri server logici ridondanti. Oppure, se individua un crash, può intraprendere la decisione di passare al cluster ridondante passivo mentre cerca di avviare il primario.

2.2.2.5 Disaster Recovery

Il disaster recovery è una proprietà che, facendo affidamento su tutte le precedenti, copre guasti di entità severamente più grande, come attacchi informatici o disastri naturali che mettono fuori uso intere strutture di server. Oltre ad offrirne ridondanza in loco, garantisce tale proprietà anche in punti geograficamente distanti e distribuiti. Ad oggi, tutti i principali fornitori di servizi cloud, garantiscono, con diversi piani e costi, l'Alta Affidabilità ed è per questo che le piu grandi aziende informatiche decidono di affifarsi a loro, preferendo avere dei costi "a consumo", ma sicuri, piuttosto che un'unica grande spesa per una infrastruttura personale.

2.2.3 IaaS

Con IaaS, abbreviazione di Infrastructure as a Service, si intende un servizio che offre una infrastruttura di macchine con un certo tipo di risorse computazionali. Ovviamente, più sono i server e le risorse richieste, più sarà l'impatto nei costi totali. Questa categoria segna il livello base di quelli che possono essere i servizi Cloud, poichè fornisce il livello di astrazione più basso. Infatti

il gestore di terze parti si "limita" a fornire le macchine (anch'esse virtualizzate come si vedrà) dando libertà nella configurazione degli applicativi, nella gestione dei sistemi operativi e del networking. Ovviamente tutto questo in Altà Affidabilità con diversi livelli e costi. Le tariffe sono a consumo, in base all'utilizzo di risorse harware nel tempo (quantità di CPU, RAM e così via).

2.2.4 PaaS

Il Platform as a Service, PaaS, è un livello di servizio che si basa su IaaS ma che astrae, oltre all'infrastruttura sottostante, anche l'eventuale middleware, sistema operativo, archiviazione e infrastruttura di rete. Cosi facendo lo sviluppatore è libero di concentrarsi maggiormente sull'applicativo, sapendo che basterà avviarlo su tale piattaforma ed il fornitore monitorerà e gestirà per lui le risorse necessare e tutta l' infrastruttura sottostante, aumentandone e diminuendone le capacità (ed i costi consecutivi) su richiesta e necessità. I vantaggi che fornisce questo livello di astrazione sono evidenti: la semplificazione e la riduzione della mole di lavoro richiesta nella gestione di un eventuale cluster in Alta Affidabilità on-premise (in locale) è notevolmente abbattuta. Permette di lavorare esclusivamente sulla qualità del codice, concentrandovi tutte gli sforzi produttivi.

2.2.5 SaaS

Il modello Software as a Service è quello con l'astrazione maggiore fra tutti. Esso propone all'utente direttamente il software richiesto attraverso l'interfaccia web, annullando tutta la complessità e gestione sottostante. A differenza del modello tradizionale dove il programmatore, o l'azienda, acquista il software e lo installa nelle varie macchine, il Cloud Provider richiede soltanto un accesso via web ad un ambiente con il software in esecuzione, già pronto all'uso per l'utente finale. Ovviamente qualsiasi gestione del carico di lavoro che ne deriva è completamente automatica e nascosta e l'Alta Disponibilità e pienamente garantita in tutti i suoi aspetti. Il Cloud Provider gestisce pure il 'versioning' garantendo sempre la versione desiderata se richiesta, oppure, l'ultima uscita sul mercato. Questo modello garantisce un costo veramente ridotto ed in genere il metodo di pagamento è un semplice abbonamento mensile o annuale, al contrario dell'acquisto della licenza che avviene tradizionalmente.

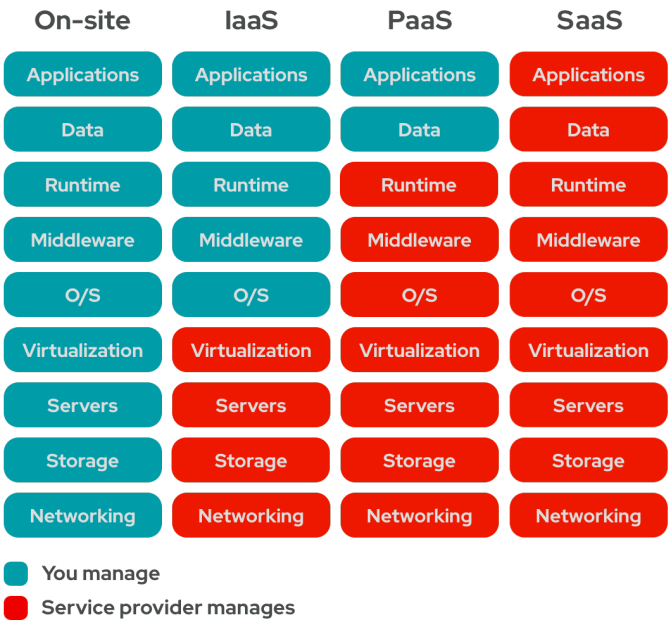

Figura 2.2: Grafico riassuntivo IaaS, PaaS e SaaS [3]

2.2.6 CaaS

Il modello Container as a Service (CaaS) rappresenta un livello di astrazione intermedio tra Infrastructure as a Service (IaaS) e Platform as a Service (PaaS). In questo modello, il cloud provider fornisce un'infrastruttura per il deployment e la gestione dei container, come Kubernetes. L'utente può così concentrarsi esclusivamente sullo sviluppo dell'applicazione containerizzata, senza doversi preoccupare della scalabilità, del networking o della gestione dell'infrastruttura sottostante, gestiti dal Provider. Grazie a questo modello lo sviluppatore può sfruttare il vantaggio del modello IaaS con la maggior flessibilità e portabilità che ha un'applicazione containerizzata, potendone trasportare il container da un ambiente Cloud ad un altro facilmente. Potendovi aggiungere successivamente un load balancing e/o uno scaling immediato attraverso la replica e la ridondanza utilizzando la stessa immagine di base. CaaS rimane uno dei servizi più potenti e flessibili offerto dai Cloud Provider.

Capitolo 3

Definizione ed Obiettivi del Sistema

Nel capitolo precedente sono stati introdotte ed approfondite, senza scendere eccessivamente nei dettagli, le definizioni di Container, Ambienti Serverless e Cloud. In questo capitolo si parlerà dei principali problemi che gli studenti affrontano per la prima volta nello sviluppo e nel testing di applicazioni containerizzate. Verrà fatta un analisi dei requisiti che una soluzione deve avere e verranno posti degli obiettivi da seguire, fino alla realizzazione di una struttura vera e propria.

3.1 Lo scoglio implementativo

Durante gli anni di Università, lo studente fa un percorso abbastanza lineare in quel che è l'apprendimento dei linguaggi, degli algoritmi e della logica matematica necessaria al fine di diventare un buon programmatore novizio. Si parte dai primi programmini in C sino ad arrivare alle complesse strutture e Design Patterns del Object Oriented Programming passando per lo studio degli algoritmi piu famosi e le loro complessità. Sin qui questo percorso è parecchio formativo e per niente facile, considerando che si sono raggiunte le conoscenze necessarie per creare un buon applicativo. La curva di apprendimento subirà un importante impennata quando, oltre a come fare correttamente un applicazione, si inizierà ad affrontare la struttura sottostante di macchine, nodi, router, configurazioni, reti, protocolli di comunicazione dei sistemi reali e del loro funzionamento. Per non parlare poi della sicurezza informatica che queste componenti dovrebbero avere. Il modello del container contiene già diversi di questi aspetti poichè racchiude in un solo elemento,

oltre alla conoscenza di come sviluppare un applicativo, numerosi concetti informatici, alcuni già citati e altri no, in particolare:

- Una solida conoscenza dei sistemi operativi, delle loro funzionalità, librerie, architetture, file system e variabili d'ambiente. Come già accennato, ogni container include al suo interno un sistema operativo, che deve essere configurato correttamente attraverso la gestione dei layer nell'immagine;
- Competenza in reti e protocolli, con particolare attenzione all'uso di porte, socket e indirizzi IP, che consentono ai container di comunicare tra loro come se fossero macchine distinte all'interno di un'infrastruttura distribuita;
- Familiarità con Docker, il suo engine e la sua CLI, che permettono di creare e gestire immagini containerizzate, visualizzare gli output, e analizzare i log per individuare eventuali errori, avvisi o comportamenti anomali;
- Esperienza con i registry, in particolare DockerHub, e la capacità di utilizzare immagini ufficiali, configurare applicazioni containerizzate tramite variabili d'ambiente o file di configurazione.

Tutto questo solo per imparare ad usare i container Docker e per cominciare a formare i primi sistemi formati da piu applicativi containerizzati. Proprio in questa fase, alcuni studenti potrebbero iniziare a subire importanti difficoltà dovute alla grossa richiesta hardware che un progetto con diversi componenti e applicativi potrebbe avere, soprattutto, se il progetto è una pipeline di "data ingestion", "data processing" e "data visualization".

3.1.1 Data Ingestion

Secondo una definizione trovata sul Web, "Un dato è una rappresentazione oggettiva e non interpretata della realtà, ciò che è immediatamente presente alla conoscenza", la differenza invece con l'informazione è netta poichè quest'ultima è una visione con un significato, generata dalla interpretazione di tali dati. Nel mondo odierno dell'Information Technology ormai tutto è trascritto in dati, a partire dal sensore di una dinamo che rileva la velocità di una ruota fino ad arrivare ai commenti di qualche complottista su un social, il flusso dei dati di qualsiasi natura e origine permea e scorre intorno a noi. Con Data Ingestion si intende appunto l'ingestione, la raccolta, di questi dati, da una fonte qualunque, come potrebbe essere una piccola stazione meteo

che misura il vento e l'umidità nel tempo o come un grosso evento automobilistico con mille sensori che segnano tempi e velocità. Questo ammontare infinito di dati che fluiscono dalla fonte, vengono raccolti attraverso software appositi come, per citarne due, Logstash o Fluentd e vengono immagazzinati ed anche organizzati in appositi spazi, pronti per essere presi ed usati per tirarne fuori qualcosa di nuovo.

3.1.2 Data Processing e Data Visualization

Il Data Processing, o elaborazione dei dati, si riferisce alla fase in cui i dati acquisiti e memorizzati dai sistemi di data ingestion vengono raccolti e sottoposti a vari processi di elaborazione. Questi dati possono essere trasformati in molteplici modi, a seconda del tipo di informazioni che si vuole estrarre o delle previsioni che si desidera ottenere. Gli strumenti principali per l'analisi e la generazione di nuove informazioni sono gli algoritmi di machine learning, che spaziano da semplici modelli di regressione lineare fino a complessi algoritmi di sentiment analysis. L'esecuzione di questi algoritmi richiede risorse hardware considerevoli, variando da alcune centinaia di megabyte di memoria RAM fino a decine di gigabyte, a seconda della complessità dell'algoritmo e della quantità di dati trattati. I dati elaborati, che possono essere prodotti in modalità batch o in tempo reale, vengono quindi nuovamente memorizzati, pronti per essere ulteriormente analizzati o utilizzati in altri processi.
La fase di Data Visualization, come suggerisce il nome, si occupa di prendere i dati e le informazioni nuove generate ed elaborate nella fase precedente e visualizzarle attraverso grafici a torta, a barre e chi più ne ha più ne metta. I software piu usati in questo campo sono Kibana e Grafana.

3.2 Risorse e limiti

Questo trio forma una pipeline importantissima, che rispecchia quelle usate da tutte le piu grandi aziente tech, a partire dall'analisi del dato dei log di qualche sistema fino alle vere e proprie analisi che il mercato compie servendosi di dati di ogni tipo, forma o natura. Gli studenti che studiano questo sistema e che devono replicarlo in un progetto personale, in una scala ovviamente più piccola della realtà, si troveranno in difficoltà dato la grande quantità di risorse che le applicazioni containerizzate richiedono, soprattutto se queste devono svolgere l'allenamento un modello di machine learning. Sarebbe un grande aiuto trovare una soluzione che possa mitigare questi problemi e fornire in maniera centralizzata e in remoto, la possibilità di avere maggiori risorse hardware e di poter testare, in un ambiente distribuito e

meglio organizzato, un deployment di container personale. Questo è l'obiettivo primario del progetto PACKS, una piattaforma di gestione e testing di container centralizzata.

3.3 PACKS, requisiti e obiettivi

Per risolvere questi problemi, PACKS, dal punto di vista infrastrutturale, si ispira ad un ambiente serverless di tipo Container as a Service. Essa vuole fornire una semplice interfaccia che permette l'avvio e la gestione dei container da parte dello studente, astraendo tutta la gestione dell'infrastruttura sottostante. I principali cloud provider forniscono un servizio CaaS basata su Kubernetes, la piattaforma open source più grande per il deployment di strutture containerizzate e per la gestione della loro scalabilità e dell'affidabilità. Più avanti si tratterà nel dettaglio questo argomento, analizzandone le componenti principali ed il loro funzionamento. Di seguito si tratteranno e si definiranno i principali requisiti funzionali e non funzionali.

3.3.1 Requisiti funzionali

Per la sua efficienza e, al tempo stesso, per le sue funzionalità, Kubernetes dovrà essere la base infrastrutturale del progetto PACKS, colei che farà da deployment orchestrator dei vari progetti creati dagli studenti. Allo stesso modo PACKS si pone come obiettivo quello di astrarre e di nascondere la complessità di tale piattaforma come, se non di più, alcuni dei principali cloud provider. Lo studente deve preoccuparsi soltanto della definizione delle immagini delle proprie componenti, evitando lo scoglio implementativo e conoscitivo del rilascio su Kubernetes. Quindi vi sarà necessario un ulteriore livello di astrazione, che possa dare un template, unico ed estremamente semplificato, adatto a tutti gli studenti. Lo studente deve essere in grado di predefinire in un unico file dichiarativo l'insieme dei container del suo progetto, specificandone i dettagli come porte usate, comandi e tutto quello che puo richiedere una immagine che segue lo standard OCI.
Per analizzare il comportamento di un container, bisogna essere in grado di leggerne e capirne l'output ed i log. PACKS deve garantire anche questo, specialmente nelle componenti in esecuzione, permettendo, in un click, quello che si farebbe attraverso diversi comandi da CLI.
Spesso è necessario collegarsi alle porte esposte di un applicativo containerizzato per la visualizzazione via browser di una interfaccia web, un esempio sono Kibana e Grafana. Quindi questa possibilità deve essere garantita anche se l'esecuzione è serverless, quindi in remoto. Tuttavia, il progetto deve

essere strutturato in modo che ogni studente sia in grado di visualizzare e gestire solo il proprio deployment senza poter alterare o manomettere quello altrui, in pratica, è requisito fondamentale un sistema di account che si basa sul login con credenziali private.
Un progetto che si ritiene finito, funzionante e pronto per la consegna, dovrebbe essere facilmente testabile e giudicabile dal docente responsabile del corso universitario. Proprio per questo, PACKS dovrebbe implementare anche un sistema di consegna affidabile che permetta all'insegnante di controllare e giudicare il risultato finale in un ambiente separato da quello dello studente.
Tutti questi requisiti dovrebbero essere sufficienti per garantire una base di partenza solida e funzionale per un buon sistema.

3.3.2 Requisiti non funzionali

Fra i requisiti non funzionali essenziali deve esservi necessariamente la possibilità di gestire più studenti in contemporanea e la capacità hardware centrale sottostante deve essere in grado di reggere le risorse richieste da una molteplicità di deployment, eseguiti contemporaneamente. Per evitare un abuso, volontario o involontario che sia, di memoria o della cpu, sarà necessaria la presenza di sistemi di limitazione di risorse per utente, cosi da implementarne la disponibilità e non intaccarne la responsività generale.
Data la possibilità di caricare un file "pacchettizzato" da parte dello studente, bisogna che vi siano degli accertamenti e dei controlli sul tipo di file caricato e la sua validità, assegnandovi anche un limite massimo in dimensione. Tali limitazioni sono obbligatorie per garantire una base di affidabilità contro potenziali malfuzionamenti derivanti da file incorretti o veri propri attacchi mirati alla negazione del servizio. Purtroppo l'infrastruttura centrale sottostante, come vedremo, non garantisce inizialmente capacità e risorse come quelle di un ambiente CaaS di un Cloud Service Provider come Google o AWS di Amazon, quindi, nelle prime fasi, è importante saper organizzare e rendere efficiente la suddivisione delle risorse per utente, cercando, al tempo stesso, di prevenire le problematiche descritte.
Per assicurare la presenza esclusiva di un account per studente ed evitare la possibilità di crearne molteplici, è essenziale usare credenziali universitarie, fornite unicamente alla registrazione ed iscrizione ad un corso universitario, come il classico codice fiscale ed il PIN.
Nel capitoli successivi verranno illustrate ed approfondite le scelte architetturali ed il design di PACKS, ma non prima di aver definito ed approfondito come si deve Kubernetes ed Helm, il suo gestore di pacchetti.

Capitolo 4

Kubernetes ed Helm

In questo capitolo verranno analizzate due componenti fondamentali del progetto PACKS, in particolare elencando le cause ed i motivi che hanno portato a queste scelte implementative.

4.1 Kubernetes

Kubernetes è uno dei sistemi per la gestione e l'orchestrazione di applicazioni containerizzate più grande mai creato, sicuramente il più importante. Sviluppato inizialmente da Google e rilasciato nel 2014 come progetto open source, diventerà velocemente uno standard ed un importantissimo punto di riferimento per tutti gli ambienti serverless e cloud moderni, che ne adotteranno le funzionalità e ne implementaranno vari servizi. Oggi giorno i suoi casi d'uso sono vasti, si passa da poter creare un'infrastruttura di vecchi laptop che si trovano in casa fino alla generazione e gestione di centinaia di server di un ambiente industriale.
Un cluster Kubernetes, essenzialmente, consiste in diverse macchine server collegate via rete, dette "Nodi". Uno di questi, chiamato "Control Plane", si occupa del monitoring e della gestione degli altri attraverso una serie di componenti interne. Il resto dei nodi, invece, sono "Nodi Worker" e sono, appunto, i server che ospiteranno i vari tipi di sistemi distribuiti contenenti applicazioni containerizzate ed interfaccie di networking.

4.1.1 Control Plane

Il Control Plane è il responsabile, nonché controllore e gestore, di tutti gli elementi e le funzionalità. Si occupa dello scheduling, dello scaling, del failover e del load balancing sui nodi Worker. Esso controlla e verifica sistemica-

mente che le componenti "Pod" rispettino costantemente un configurazione preimpostata attraverso un file "yaml". Questo tipo di programmazione si chiama "configurazione dichiarativa" poichè, appunto, si dichiara prima lo "status ideale" che la piattaforma deve raggiungere e mantenere. Il Control Plane è composto a sua volta da componenti, di cui le principali:

- **etcd** è il database permanente del cluster, in esso sono salvate le configurazioni e gli stati. Garantisce fault tolerance attraverso un sistema di ridondanza e failover su altri nodi. Si tratta essenzialmente di un database "chiave-valore" che comunica maggiormente con l'interfaccia "kube-apiserver" attraverso richieste e risposte HTTP di informazioni in formato prevalentemente JSON. Le prestazioni di questa componente sono al top dato che garantisce una affidabilità elevata grazie ad una ridondanza sincronizzata fra i nodi gestita da un protocollo di sincronizzazione di nome "Raft".

- Il **kube-scheduler**, come suggerisce il nome, si occupa della schedulazione dei vari componenti Kubernetes (Pod) tra i nodi disponibili, scegliendo quello più adatto in base a una serie di parametri. Quando un Pod non è ancora stato allocato su un nodo, il Kube Scheduler avvia un processo di selezione suddiviso in più fasi. In primo luogo, esegue un filtraggio dei nodi disponibili, escludendo quelli che non dispongono di risorse sufficienti, quelli con un'architettura incompatibile o che non soddisfano specifici requisiti.
  Dopo questa fase di filtraggio, i nodi rimanenti vengono ordinati e classificati in base a vari criteri, come la quantità di risorse disponibili o la vicinanza a volumi di storage necessari. Il nodo che ottiene il punteggio migliore viene scelto per eseguire il Pod. Questo meccanismo è cruciale per garantire un efficiente bilanciamento del carico tra le risorse del cluster, rendendolo essenziale soprattutto in ambienti con numerosi nodi worker.

- il **kube-controller-manager** è la componente del control plane che si occupa, appunto, di mantenere ed eseguire i controller. Questi sono "controllori" dello stato di un certo tipo di componente e delle sue proprietà. Verificano continuamente la situazione in cui versa il cluster e, se questa non corrisponde a quella desiderata dall'utente o dalle configurazioni principali, ne compie tutti i cambiamenti e le modifiche necessarie per ritornarvici.
  In particolare vi sono controller per il numero di repliche (o ridondanze) di un Pod, per lo stato di salute dei nodi, per lo stato di completamento ed avanzamento dei Job Kubernetes, per il collegamento dei Service ai

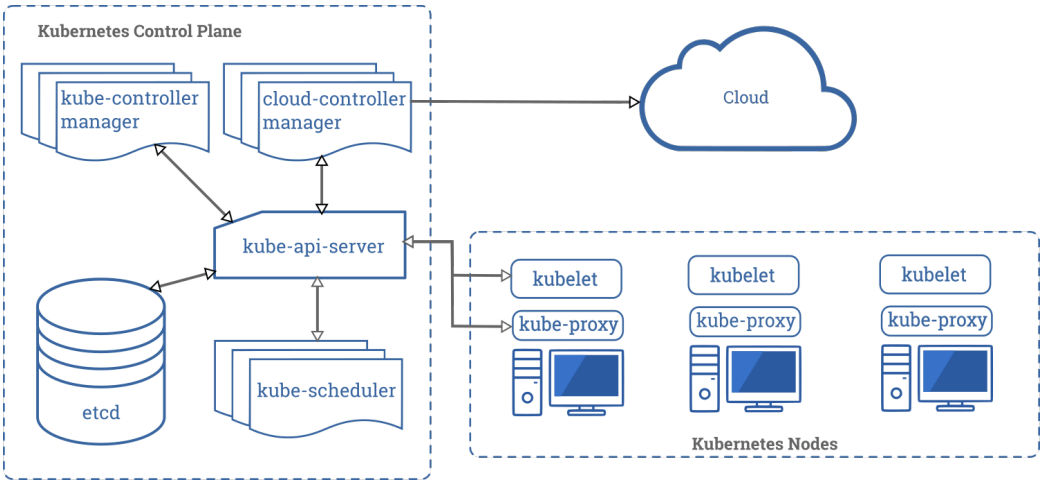

Figura 4.1: Componenti Control Plane [4]

Pod corretti e così via. In genere in un cluster solo una istanza di questa componente è attiva ma è possibile settarne delle ridondanze di tipo Attivo-Passivo.

- il **cloud-controller-manager** è un controller che inizialmente faceva parte del kube-controller-manager ma che successivamente, con la sempre maggior integrazione con l'ambiente ed i servizi Cloud, venne scorporato e fatto un elemento a se. Come il nome lascia intendere, si occupa di interfacciarsi e gestire le risorse e gli elementi Kubernetes stanziati in remoto.

- L'**API server** è l'interfaccia principale del Control Plane di Kubernetes, che espone tutte le API necessarie per ricevere comandi, template e configurazioni dichiarative, e si occupa della loro esecuzione. Funziona anche come mediatore tra le varie componenti del sistema, controllandone lo stato e i compiti principali. Quando riceve richieste, l'API server raccoglie i dati interrogando le parti necessarie e inoltra le risposte al destinatario. Inoltre, gestisce la lettura e scrittura in etcd, memorizzando ed estraendo informazioni e configurazioni di tutto il cluster. Si assicura che le regole di sicurezza vengano applicate correttamente e che vengano rispettati i limiti imposti. In definitiva, l'API server può essere considerato il cuore di Kubernetes, essendo responsabile del coordinamento e del funzionamento di tutte le sue funzionalità.

4.1.2 Nodi Worker

Ogni nodo, che sia Worker o Control Plane, deve aver installate e configurate degli applicativi fondamentali, nel dettaglio:

- **Kubelet** è la componente che si occupa di avviare fisicamente e gestire i Pod sul nodo in questione. Essa è in costante comunicazione con l'API Server. Quest'ultimo, attraverso lo scheduler, ordina all'istanza Kubelet nel nodo scelto, di scaricarne l'immagine, montarne i volumi e di avviare, eliminare o ricreare un Pod. In particolare, se questo ha un crash o si interrompe, tenta di riavviarlo. Tale componente informa costantemente l'API Server dello stato dei Pod che esegue ed in particolare ne distungue quelli interni.
  Un altro importantissimo compito svolto è la gestione dei volumi allocati nei container e dei control groups (cgroups) Linux dei container, assegnandovi inoltre le risorse hardware necessarie alla sua esecuzione. Inoltre, come un controller, confronta constantemente lo stato interno dei container con quello desiderato ed attua le corrette azioni per mantenerlo.

- **CRI** o "Container Runtime Interface" è l'applicativo responsabile dell'esecuzione dei container. Esso è fondamentale ed attraverso una sua API comunica costantemente con Kubelet. In pratica, il CRI vi si può considerare come una componente completamente integrata. Vi sono varie implementazioni e versioni supportate da Kubernetes, le principali sono: "containerd", "dockerd" e "cri-o".

- **kube-proxy** è una componente fondamentale per la gestione del traffico di rete e del routing all'interno del cluster. Come suggerisce il nome, si tratta di un vero e proprio proxy che riceve i pacchetti e sceglie a quale Pod inoltrarli. Configura l'instradamento servendosi delle regole di "iptables" o "IPVS". Il primo è un software che gestisce il routing e le regole di firewall a livello ip, il secondo è un alternativa a iptables, più efficiente in ambienti con un grande traffico di pacchetti. In base all'interfaccia di networking che l'utente associa al Pod, kube-proxy ne modifica le regole di instradamento ed il comportamento. Vari oggetti Kubernetes come i Service e gli Ingress ne stabiliscono e modificano la configurazione interna, si vedrà più avanti nel dettaglio come.
  Se un Pod dovesse risultare non correttamente in funzione, sarà escluso dalle regole di routing e non raggiungibile fin quando non tornerà ad uno stato "Ready".

- **DNS**, o "Domain Name System", componente che ha la tipica funzione di un classico server DNS, nella rete interna al cluster. Ovvero, grazie ad esso avviene la risoluzione dei nomi host, in particolare quelli dei Service, in indirizzi ip interni a Kubernetes.

4.1.3 Oggetti e Manifest Kubernetes

Nei paragrafi precedenti si è discusso dell'infrastruttura di base del cluster e delle funzionalità delle componenti. Adesso, per iniziare ad interagirvi ed a rilasciare in questo ambiente distribuito i container desiderati, bisogna affrontare nel dettaglio quelli che sono gli "Oggetti" Kubernetes che formano quest'importante ecosistema. Fra i principali vi sono: Pod, Deployment, Statefulset, Daemonset, Jobs, Service, Ingress, Persistent Volumes e Persistent Volume Claim.

4.1.3.1 Manifest

Prima di tutto bisogna sapere come si descrive e si definisce un Oggetto Kubernetes. Un **manifest** è nientemeno che un file di configurazione scritto in YAML (o anche JSON) usando un linguaggio di configurazione dichiarativa, ovvero, vengono definite tutte le componenti e variabili che rappresentano lo stato finale desiderato dell'oggetto e che Kubernetes cercherà sempre di raggiungere.

Codice 4.1: esempio manifest yaml di un Pod con Nginx

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: nginx-proxy-pod
5   labels:
6     app: nginx-proxy
7 spec:
8   containers:
9   - name: nginx-proxy
10    image: nginx:latest
11    ports:
12    - containerPort: 80
13    volumeMounts:
14    - name: nginx-config-volume
15      mountPath: /etc/nginx/nginx.conf
16      subPath: nginx.conf
17  volumes:
18  - name: nginx-config-volume
19    configMap:
20      name: nginx-config
```

Come è possibile vedere dal manifest dell'esempio Codice 4.1, tutti gli oggetti, nella loro definizione, seguono una struttura predefinita, in particolare:

- **apiVersion** è il campo nel quale viene specificata la versione della API Kubernetes a cui il manifest deve far riferimento. In base a questo valore possiamo specificare e descrivere diversi Oggetti. Nell'esempio viene utilizzata la versione 'v1';
- **kind**, campo che specifica il tipo di oggetto definibile dalla versione API selezionata, in questo caso viene indicato che si tratta di un Pod;
- **metadata**: questa sezione specifica il "name" e le "labels" dell'oggetto, fornendo un identificativo univoco. Le "labels" sono particolarmente utili poiché permettono ad altri oggetti di fare riferimento a uno specifico elemento o a un gruppo di elementi. In dettaglio, altri oggetti possono utilizzarle per effettuare delle vere e proprie query, tramite dei selettori, per individuare e interagire con un oggetto specifico o un gruppo di oggetti che condividono determinate etichette.
- **spec**, vasta sezione con diversi sottocampi, in base al tipo di Oggetto. Come si nota nell'esempio 4.1, in genere vengono elencate e descritte le componenti inerenti alla configurazione ed il loro stato desiderato.

4.1.3.2 Pod

Questo oggetto è gia stato citato precedentemente in più esempi, ma mai approfondito. "I Pod sono la più piccola unità in Kubernetes di cui possiamo fare il deployment" [5]. Rappresenta a tutti gli effetti un server con uno o più container in esecuzione, infatti questi avranno lo stesso hostname e lo stesso indirizzo Ip nella subnetwork del cluster, potranno comunicare attraverso sistemi di Inter Process Communication e leggere/scrivere gli stessi file dallo storage in comune dell'Oggetto. Quindi si puo pensare ad esso come un vero e proprio server logico con diversi applicativi containerizzati.
Però, creare un Pod multicontainer è sconsigliato e va contro le proprietà di granularità, scalabilità e ridondanza, perchè questi saranno, per costruzione, sempre schedulati ed avviati tutti nello stesso nodo. Se fra gli applicativi vi è un servizio che viene scalato orizzonatalmente, ridondando il Pod, se ne replicherà tutto l'insieme, creando duplicati di applicativi non richesti, occupando risorse hardware inutilmente e sottraendole alle componenti che veramente ne hanno bisogno. Infatti conviene sempre creare un Pod per container ove possibile cosicché non possa venirne intaccata l'affidabilità generale.
Nell'esempio 4.1 vediamo il manifest di un Pod che contiene un container

con Nginx. Nel dettaglio, dentro il campo "spec" vi sono due campi principali: "containers" e "volumes". Nel primo, oltre al nome, vi si definisce l'immagine che Kubelet ricercherà nei principali registry (in genere DockerHub), le porte da esporre ed il protocollo che accettano, eventuali variabili d'ambiente, "liveness probe", "readiness probe", limiti di risorse da utilizzare e "volumeMounts".
Il **liveness probe** è il set di comandi da eseguire per ricavare e definire lo stato del container in modo personalizzato, in particolare, se esso è attivo e funzionante. Vi sono diversi metodi e comandi per testare la risposta di un applicativo, tra questi il più comune è una richiesta HTTP GET ad un indirizzo particolare. Il comando viene eseguito ad ogni frazione di tempo definita dall'utente e, fin quando si riceve una risposta positiva, lo stato verrà considerato come attivo ed in salute.
La **readiness probe**, cosi come la "liveness probe", è un set di comandi che indica lo stato del container, in particolare però all'avvio o al reset, quando deve divenire 'Ready'. Il set di comandi non varia molto dal precedente ed in genere anche qui la GET HTTP è privilegiata.
Entrambi formano l'HealthCheck, come si evince nel codice 4.2, si puo decidere nei minimi dettagli il comportamento del test, potendone settare il tempo iniziale di attesa (initialDelay), ogni quanto ripetere il test con "periodSeconds", quanto tempo aspettare per una risposta con "timeoutSeconds", decidere dopo quanti tentativi falliti si considera non funzionante il servizio con "failureThreshold" e (specifico per readinesProbe) il campo "successThreshold" che indica dopo quanti successi bisogna innescarne lo stato "Ready".

Codice 4.2: esempio Healthcheck

```
livenessProbe:
    httpGet:
        path: /healthy
        port: 8090
    initialDelaySeconds: 10
    timeoutSeconds: 3
    periodSeconds: 10
    failureThreshold: 3
```

Attraverso la l'elemento "resources", è possibile settare i limiti massimi e minimi delle risorse hardware, in particolare quello che concerne l'utilizzo della CPU e della memoria centrale. Un codice esempio è il 4.3

Codice 4.3: esempio Resources

```
resources:
    requests:
        cpu: "500m"
```

```
4           memory: "256Mi"
5       limits:
6           cpu:"900m"
7           memory:"500Mi"
```

Come si può notare la quantità di CPU usata è misurata in "mi", o "Millicores". "1000mi" indicano il 100% di un core della CPU, quindi nell'esempio 4.3 il manifest indica uno stato minimo libero del 50% di un core CPU, 256 Megabyte di memoria RAM ed un massimo occupabile di un core intero e 500 Megabyte di RAM.
Una risorsa fondamentale che può essere specificata all'interno del campo "spec" sono i **VolumeMounts**. Essi permettono di specificare il volume di memoria da montare in uno specifico percorso all'interno del Pod.

Codice 4.4: esempio volumeMounts

```
1 volumeMounts:
2     - name: env
3       mountPath: /kluster-ui/.env
4       subPath: .env
```

In particolare, il nome seleziona fra i tipi di volumi, quello da montare seguendo il percorso indicato in "mountPath". Opzionalmente è possibile specificare un singolo file o percorso all'interno del volume con l'uso di "subPath".

Oltre alla definizione di "containers", in "spec" è possibile inserirvi i **Volumes**. Essi rappresentano sezioni di memoria che possono essere create da cartelle presenti nella macchina host, da **Persistent Volume Claims** o da **ConfigMaps**.

4.1.3.3 Namespace

I **namespace** Kubernetes possono essere intesi come dei contenitori logici che suddividono i vari oggetti come i Pod, i Service ecc... Tutti gli elementi che fanno parte dello stesso namespace possono comunicare facilmente tra di loro grazie ad una risoluzione facilitata del nome host dal server DNS (si approfondirà più in avanti). In genere è proprio grazie ad essi che un grosso ambiente di sviluppo può suddividere il proprio software in namespace "dev" oppure "produzione". Entrambi potrebbero coesistere tranquillamente e l'uno è totalmente separato dall'altro. Kubernetes, fra le varie funzionalità, offre un accesso "Role Based" via credenziali alle varie componenti, fra queste anche quelle sotto un certo namespace.
Inoltre alcune risorse possono essere visibili soltanto all'interno di questi contenitori logici e nascoste all'esterno fornendo un certo livello di sicurezza. Alla creazione di un oggetto, se non specificato esplicitamente nel manifest, esso

verrà assegnato nel namespace "default". Le componenti interne del cluster si trovano invece all'interno di "kube-system".

4.1.3.4 ReplicaSet

Un ReplicaSet è un oggetto che si occupa di controllare il numero di Pod in esecuzione e di mantenerlo in base all'attributo "replicas" appositamente settato nel manifest. La selezione dei Pod a cui applicare tali regole avviene attraverso dei selettori di "labels" (metadata descritti precedentemente). Una volta trovati i Pod questi verranno scalati orizzontalmente fino a raggiungere il numero desiderato. Questo vale anche se uno di questi si interrompe in modo anomalo, infatti. in tal caso, verrà rischedulato il necessario per ritornare alle quantità normali.
Il suo compito è molto simile al Controller del control-plane che gestisce le repliche. In effetti si può affermare che si tratti di una sua versione migliorata, in particolare nei selettori che permettono query più complesse.

Codice 4.5: esempio di manifest ReplicaSet

```
1 apiVersion: apps/v1
2 kind: ReplicaSet
3 metadata:
4   name: nginx-replicaset
5 spec:
6   replicas: 3
7   selector:
8     matchLabels:
9       app: nginx
10  template:
11    metadata:
12      labels:
13        app: nginx
14    spec:
15      containers:
16      - name: nginx
17        image: nginx:1.21
18        ports:
19        - containerPort: 80
```

Come si puo notare nell'esempio 4.5, sotto "spec", vi si trovano il campo chiave "replicas" già citato, il "selector" con "matchLabels" ed i "label" da selezionare, la voce "template" dove, appunto, viene definito il template del Pod da replicare.

4.1.3.5 HPA

L' Horizontal Pod Autoscaler, come il nome stesso suggerisce, si tratta di un Oggetto che, come il ReplicaSet, gestisce le repliche di un Pod ma, invece di settarne manualmente il numero, lo gestisce in maniera automatica, in base alla risorse occupate. In particolare vi si può settare un target di risorse occupate dove al di sotto viene diminuito il numero di Pod nei nodi e sopra, invece, viene aumentato.

4.1.3.6 Deployment

I Deployments sono oggetti Kubernetes che si basano sui ReplicaSet per gestire le repliche dei Pod, ma offrono funzionalità avanzate per la gestione e l'aggiornamento delle applicazioni. Una delle principali caratteristiche introdotte dai Deployments è la possibilità di gestire aggiornamenti in modo strategico, come i "rolling updates". Questo meccanismo consente di distribuire nuove versioni dell'applicazione progressivamente, senza mai ridurre il numero di Pod attivi sotto il valore desiderato. Durante il processo di aggiornamento, un nuovo Pod con la versione aggiornata viene avviato, e solo quando è in stato 'Ready', un Pod della vecchia versione viene terminato ed eliminato dal cluster. Questo ciclo continua fino a quando tutti i Pod sono stati sostituiti con la nuova versione.
Un altro vantaggio chiave dei Deployments è la possibilità di eseguire una strategia di tipo "rollback", ovvero il ritorno a una versione precedente dell'applicazione. Le versioni precedenti non vengono eliminate completamente, ma rimangono memorizzate, permettendo di tornare rapidamente alla vecchia configurazione in caso di problemi con l'aggiornamento. Si potrebbe considerare questo processo come l'inverso del "rolling update", poiché ripristina gradualmente la versione precedente.
Oltre al rolling update, esiste anche la strategia di aggiornamento "recreate", che è più drastica. In questo caso, Kubernetes termina tutti i Pod della vecchia versione prima di avviare quelli della nuova versione. Sebbene questa modalità causi un'interruzione temporanea del servizio, è utile quando la vecchia e la nuova versione dell'applicazione non possono coesistere, ad esempio a causa di incompatibilità o conflitti.
L'uso dei Deployments è particolarmente indicato per il mantenimento a lungo termine delle applicazioni, soprattutto negli ambienti di produzione aziendale, dove l'affidabilità, la continuità del servizio e la capacità di gestire aggiornamenti senza interruzioni sono di vitale importanza.

Codice 4.6: esempio di manifest Deployment, dal progetto PACKS

```
1 apiVersion: apps/v1
```

```
2 kind: Deployment
3 metadata:
4   name: kluster-ui-dp
5 spec:
6   replicas: 1
7   selector:
8     matchLabels:
9       app: kluster-ui-dp
10  template:
11    metadata:
12      labels:
13        app: kluster-ui-dp
14    spec:
15      containers:
16        - name: kluster-ui-dp
17          image: giuseppebonanno99/klusterui:latest
18          ports:
19            - containerPort: 3000
20          volumeMounts:
21            - name: env
22              mountPath: /kluster-ui/.env
23              subPath: .env
24
25      volumes:
26        - name: env
27          configMap:
28            name: env-cfmap
29            items:
30              - key: .env
31                path: .env
```

Dal codice 4.6 si può osservare il manifest di un Deployment usato in PACKS. Il primo campo "spec" specifica il ReplicaSet che, a sua volta, definisce il Pod ed il numero di repliche. La configurazione più esterna, riguardante la risorsa Deployment, attraverso i selettori, referenzia e gestisce i ReplicaSet indicati nella query.

4.1.3.7 ConfigMap

La ConfigMap è un tipo di risorsa particolare che permette, attraverso degli attributi chiave-valore di inserire in un ambiente, o namespace, delle configurazioni utili agli applicativi interni ai Pod. Queste configurazioni possono essere inserite sotto forma di file testuale (montati come Volumes), di variabile d'ambiente o semplicemente variabili testuali. Una ConfigMap essenzialmente serve a decentralizzare la dipendenza da configurazioni e variabili nei manifest degli Oggetti, evitando inutile hardcoding e consentendo di mantenere configurazioni diverse per ambienti diversi con stessi manifest.

Codice 4.7: esempio di manifest ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-config
data:
  database_url: "mysql://user:password@db-host:3306/mydatabase
      "
  boolean: "true"
main.py:
    import matplotlib
    from scipy.stats import norm
    #basic example of python source code
    print("example")
    return true
```

4.1.3.8 Persistent Volume

Come indicato dal nome, il **Persistent Volume** è un tipo di risorsa Kubernetes che permette di definire uno storage indipendente permanente. Il PV può essere montato su una cartella del file system o su elementi terzi come storage remoti o in Cloud. Si tratta di un elemento a se, indipendente dagli altri, infatti, se un Pod che lo usa si interrompe, il volume non scompare ma rimane intatto e pronto ad essere collegato ad un altro elemento.
Vi sono presenti alcuni modi per settare delle modalità di lettura e scrittura permesse, in generale: ReadWriteOnce, ReadOnlyMany, ReadWriteMany.

Codice 4.8: manifest PV di helm-manager

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: shared-pv
spec:
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteMany
  hostPath:
    path: /shared/uploads/
    type: DirectoryOrCreate
```

4.1.3.9 Persistent Volume Claim

Il **Persistent Volume Claim** è un elemento che si occupa di fare un "binding" fra un Persistent Volume ed un Pod. Nel momento in cui quest'ultimo

lo monta nella sua parte del file system, il PV non può essere né utilizzato né collegato ad altri oggetti. Quando un Pod si interrompe, tutti i PV collegati ad esso attraverso Persiste Volume Claims possono anche essere svuotati e resettati, se specificato nel manifest.

Codice 4.9: manifest PVC di helm-manager

```
1 apiVersion: v1
2 kind: PersistentVolumeClaim
3 metadata:
4   name: shared-pvc
5 spec:
6   accessModes:
7     - ReadWriteMany
8   resources:
9     requests:
10      storage: 2Gi
```

4.1.3.10 Service

Un **Service** è un oggetto che si occupa di definire e dare un interfaccia di rete statica ad elementi interni al cluster variabili come Pod o Deployment. In particolare, se si hanno dei container con porte esposte, il Service ne può settare il comportamento ed il tipo di instradamento. Prendiamo in esempio un cluster con un Pod di una WebApp che ascolta su porta 8080. Se il sistema sarà replicato diverse volte vi saranno diversi indirizzi IP per lo stesso tipo di servizio (pagine web). Queste potrebbero cambiare spesso semplicemente per uno scaling oppure per un malfunzionamento che ne innesca il riavvio. Un Service, attraverso il selettore, selezionerà e raccoglierà gli IP dei vari Pod replicati aggiornati, quando almeno uno di questi è in stato di 'Ready'. Avendo egli stesso un indirizzo ed un hostname statico, la comunicazione fra Pod potrà essere più affidabile, riducendo il rischio di disconnessioni per cambio IP.
Quindi, le richieste che il Service riceve vengono inoltrate, distribuendole fra i vari Pod 'Ready', principalmente seguendo una strategia di diffusione "round-robin".
VI sono diversi tipi di Service. Il "ClusterIp" è quello di default, impostato se non ne viene specificato un altro.
Il tipo "NodePort" estende il "ClusterIp" ed effettua il binding di una porta del nodo host con quella specificata nel manifest. In breve, permette di accedere ai servizi interni dall'esterno del cluster.
Il tipo "LoadBalancer", distribuisce le richieste in base al carico dei nodi ed espone ad Internet un ip pubblico. Necessita di un load balancer Cloud esterno e non ne implementa uno personale.

Il tipo "ExternalName" è un Service che fornisce un interfaccia statica ad una risorsa esterna al cluster, creando un record DNS interno.

Codice 4.10: manifest di Service per componente helm-manager (PACKS)

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: helm-manager
5 spec:
6   #type: NodePort
7   selector:
8     app: helm-manager-dp
9   ports:
10    - protocol: TCP
11      port: 9000
12      targetPort: 9000
13      #nodePort: 30000
```

L'esempio 4.10 mostra il manifest di un Service per una componente del progetto PACKS, nello specifico l'Helm-Manager. Come indicato nella sezione "ports", viene collegata la porta 9000 del Service stesso con la porta 9000 dei container selezionati tramite le etichette definite nel selettore. Se si volesse modificare il tipo di Service passando a NodePort, basterebbe specificarlo aggiungendo "type: NodePort" nel campo "spec" e definendo il numero di porta del nodo con "nodePort: numero_intero", insieme ai campi "port" e "targetPort" (come illustrato nell'esempio). Va notato che il range delle porte per i NodePort è compreso tra 30000 e 32767. Se non si specifica manualmente una porta del nodo, Kubernetes ne assegnerà una casualmente all'interno di questo intervallo.

4.1.3.11 StatefulSet e Daemonset

Sono componenti Kubernetes che gestiscono Pod, come lo è il Deployment, ma lo fanno in modi differenti per diversi casi d'uso. Lo **StatefulSet** è studiato per mantenere uno stato interno fra i Pod. L'aggiornamento avviene gradualmente, rispettando gli ordini di creazione iniziali. Quando viene associato un PV e un PVC ad un Deployment con diverse repliche, queste condividono tutte lo stesso storage. Invece nel StatefulSet ogni Pod ha il proprio PVC e mantiene uno stato a se diverso dalle sue repliche. Questo tipo di risorsa Kubernetes è molto utile per applicazioni in cui è particolarmente importante mantenere uno stato per ogni istanza.
I **Daemonset** invece, si occupano di mantenere un Pod in esecuzione in ogni nodo del cluster. Se viene aggiunta una nuova macchina Worker, esso vi schedula immediatamente il Pod a cui fa riferimento attraverso il selettore.

Molto utile quando bisogna mantenere interfacce di rete o istanze e componenti che, per funzionare, devono essere presenti in ogni nodo. Un esempio lo troviamo nel progetto PACKS, esattamente in un reverse proxy Nginx. Anche i Service hanno un comportamento specifico per i Daemonset, infatti se questi sono NodePort, le porte aperte saranno in ogni nodo del cluster.

4.1.4 CNI Plugins

Definite tutte le componenti di rete dei nodi, come i DNS e i kube-proxy, è necessario introdurre le componenti fondamentali che gestiscono fisicamente il networking. I kube-proxy, che gestiscono gli endpoint di rete dei Pod secondo le configurazioni date dai Service, lavorano in realtà a livello di Trasporto nell'infrastruttura del cluster. Vi è necessaria una implementazione dei layer di rete OSI sottostanti all'interno del cluster per metterne, a tutti gli effetti, in comunicazione i Pod. In questo senso vengono in aiuto i CNI Plugin.
I **Container Network Interface** sono plugin di rete esterni a Kubernetes e seguono uno standard open-source. Formano quello che si puo definire come lo "scheletro" della rete Kubernetes. Il plugin assegna gli indirizzi di rete univoci ai Pod in base ad una sottorete specificata e permette il routing a livello Datalink fra di essi. Le componenti che formano, scalano insieme al cluster attraverso l'uso di Daemonset e interfacce. Alcuni dei più diffusi sono Flannel e Calico.

4.1.4.1 Flannel

Flannel è considerato uno dei CNI plugin più semplici ed anche quello ampiamente più utilizzato. Si serve di una rete overlay, in particolare di "Virtual Extensible LAN" con tunneling attraverso UDP, per far comunicare i nodi non fisicamente collegati in una infrastruttura di rete. Questo di solito comporta un overhead nella trasmissione e ricezione dei pacchetti ma nulla di trascendentale. Invece se i nodi sono fisicamente appartenenti ad una infrastruttura già definita verrà usato "Host-GW" che, appunto, ne sfrutterà la rete senza l'uso di overlay, rendendo più efficiente la comunicazione.
Flannel crea per ogni nodo una subnet, con un range di indirizzi IP successivamente sfruttati dai Pod. Inoltre si affida alla componente etcd per memorizzare tutte le subnet e da lì costruire tabelle di routing funzionanti per tutti gli elementi. In fin dei conti si tratta di un plugin molto basilare e che si integra alla perfezione con Kubernetes.

4.1.4.2 Calico

Calico, a differenza di Flannel, è uno dei plugin CNI più avanzati che si possano usare, poiché ha la capacità di implementare anche policy e regole di rete oltre al routing basilare. Questo plugin usa un vero e proprio routing tra i nodi simile a quello che si trova nelle reti comuni. Oltre ad avere un "demone" su ogni nodo, il routing è gestito da una componente chiamata "Felix" che, in breve, si occupa di creare le tabelle e modificare le regole "iptables" in base alla policy predefinita.
Le policy spaziano dalla possibilità di decidere i namespace o i determinati Pod da cui accettare il traffico in entrata per un determinato IP, fino al controllo del traffico in uscita versi determinati indirizzi.
In breve, Calico offre una gestione più completa ed efficiente del routing interno al cluster.

4.2 Helm

Sebbene Kubernetes fornisca un'astrazione potente per la creazione e la gestione di sistemi distribuiti di applicazioni containerizzate, l'uso di numerose componenti e risorse può risultare complesso e poco intuitivo. La gestione di un'applicazione Kubernetes richiede la creazione di diversi file manifest per le varie componenti, rendendo lo sviluppo e l'aggiornamento continuo dell'applicazione, insieme alle sue dipendenze, un processo macchinoso. In questo contesto, **Helm** si rivela lo strumento mancante in questo intricato ecosistema, semplificando notevolmente la distribuzione e il versioning delle applicazioni Kubernetes.
Esso offre un sistema standardizzato per il packaging che ne aiuta, oltre alla distribuzione, l'adattabilità nelle varie configurazioni e personalizzazioni in base ai mille diversi casi d'uso che un utente potrebbe avere.

4.2.1 Un gestore di pacchetti

Lo si può definire un completo gestore di pacchetti, come i famosi "apt" e "pacman" di alcune distribuzioni Linux, ma specifico per applicativi Kubernetes. Un pacchetto è chiamato **Chart**. Al suo interno vi sono presenti tutti i manifest delle componenti dell'applicativo da "inscatolare" oltre a files che fungono da configurazioni, che descrivono le parti interne e che ne gestiscono il versioning. Una istanza di un Chart è chiamata **release** e ve ne possono essere diverse in ambienti o namespace differenti, che si basano sugli stessi template e, quindi, che hanno la stessa struttura interna, differendo soltanto per qualche parametro di configurazione.

In base alle modifiche apportate, il versioning del pacchetto deve essere corretto e aggiornato. Un Chart, come si vedrà, può essere una cartella o un file .tgz, in genere il secondo viene rilasciato come nuova versione del pacchetto e caricato su registry come GitHub o Artifact Hub, dal quale se ne possono scaricare diversi e da qualsiasi sviluppatore che li ha resi pubblici.

4.2.2 Struttura Chart

Prima del rilascio e durante lo sviluppo, il Chart si presenta come una struttura formata da una directory principale che contiene a sua volta subdirectory e files.

Codice 4.11: struttura Chart

```
1 my-chart/
2   - Chart.yaml
3   - values.yaml
4   - charts/
5       |- subchart_folder/
6   - templates/
7       |- _helpers.tpl
8       |- deployment.yaml
9       |- service.yaml
10      |- configmap.yaml
11      |- NOTES.txt
12  - .helmignore
```

Come si evince nella struttura esempio 4.11 vi sono diverse componenti che formano i metadati e le parti di un applicativo Kubernetes.
Chart.yaml è il file principale che identifica, descrive e contiene i metadati del Chart. Fra questi in particolare vi è il nome, la versione, una descrizione ed eventuali dipendenze.

Codice 4.12: esempio Chart.yaml

```
1 apiVersion: v2
2 name: my-chart
3 description: A Helm chart for Kubernetes
4 version: 1.0.0
5 appVersion: 1.16.0
```

La cartella "charts/" contiene, appunto, Chart da cui il principale dipende per funzionare, quindi "subChart".
Nel file **values.yaml** sono contenuti, attraverso una sintassi "chiave:valore" le eventuali configurazioni personalizzate che l'utente può sovrascrivere. Questo file viene letto da Helm nel momento della installazione del Chart, ossia quando vengono analizzati e scritti i manifest, i quali, saranno suc-

cessivamente mandati, dopo un controllo sintattico ,a Kubernetes tramite API.

Codice 4.13: esempio values.yaml

```
1 replicaCount: 2
2 image:
3   repository: nginx
4   tag: "1.16"
5   pullPolicy: IfNotPresent
6 service:
7   type: ClusterIP
8   port: 80
9
10 variabile_a_caso: 12345
11 lista1:
12 - nome: val1
13   valore: 123456677
14 - nome: val2
15   valore: 09898776
```

Dall'esempio 4.13 si nota come la sintassi non segue una regola ben precisa come i manifest ed il file Chart.yaml. Gli attributi "chiave:valore" possono essere totalmente arbitrari, l'importante è che il motore di template sappia il nome corretto della chiave per trovarne il valore associatovi.

4.2.2.1 Templates

Oltre a fungere da package manager, Helm ingloba il template engine di Go (Go templates) che permette di "templatizzare" e parametrizzare i manifest Kubernetes contenuti all'interno della cartella "/templates". Nel dettaglio, è possibile inserirvi cicli condizionali e variabili direttamente dentro i manifest. All'interno di un blocco di codice aperto e chiuso da un paio di parentesi graffe si possono usare le variabili che si trovano nel values.yaml usando l'oggetto ".Values", che le contiene tutte.
Se ad esempio si volesse inserire il valore della chiave "variabile_a_caso" all'interno di un manifest, il blocco di codice deve essere il seguente:"

```
1     {{ .Values.variabile_a_caso }}
```

Vi sono altri oggetti Helm che accedono a diverse variabili, ad esempio ".Chart" contiene i metadati relativi a Chart.yaml, ".Files" consente di accedere ad eventuali file inclusi nella cartella e ".Release" contiene informazioni sull'instanza del Chart avviata.
Simile ad un linguaggio di programmazione, il template engine supporta blocchi di controlli condizionali (if/else), cicli e funzioni.
Il blocco condizionale è espresso come segue

```
1     {{- if .Values.immagine }}
2     image: {{ .Values.immagine }}:{{ .Values.immagine.versione
          }}
3     {{- end }}
```

Se è presente e definita in values.yaml la chiave "immagine", allora verrà inserito "image: valore_immagine:valore_versione" nel manifest. Invece se tra le variabili è presente una lista, come "lista1" nell'esempio 4.13, attraverso un ciclo "range" è possibile iterarne gli elementi.

```
1 env:
2 {{- range .Values.lista1 }}
3   - name: {{ .nome }}
4     value: {{ .valore }}
5 {{- end }}
```

La "pipeline" è una concatenazione di funzioni che ricevono in input un valore e ne danno in output un altro. Esse sono divise e serializzate, in base all'ordine di esecuzione, da un carattere visibile dagli esempi.
Le funzioni disponibili sono tante e diverse, sono oltretutto consultabili online, nella documentazione ufficiale Helm. Alcune di queste sono:

- "lower" e "upper", convertono i caratteri di una stringa in minuscolo se questi sono in maiuscolo (lower) e viceversa (upper);
- "default" 'valore_default', inserisce il valore all'interno della stringa successiva se quella passatagli dalla pipeline è nulla;
- "len", conta il numero di elementi di una lista;
- "quote" e "squote", rispettivamente, una ingloba la stringa in delle virgolette e l'altra la estrae da esse se presenti;
- "trim", rimuove spazi vuoti inutili;

Codice 4.14: esempio completo template di un Deployment

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: {{ .Release.Name }}-deployment
5 spec:
6   replicas: {{ .Values.replicaCount }}
7   selector:
8     matchLabels:
9       app: {{ .Release.Name }}
10  template:
11    metadata:
```

```
12      labels:
13        app: {{ .Release.Name }}
14    spec:
15      containers:
16        - name: {{ .Chart.Name }}
17          image: {{ .Values.image.repository }}:{{ .Values.
                image.tag }}
18          ports:
19            - containerPort: {{ .Values.service.port }}
20          resources:
21            limits:
22              memory: {{ .Values.resources.limits.memory }}
23              cpu: {{ .Values.resources.limits.cpu }}
24            requests:
25              memory: {{ .Values.resources.requests.memory }}
26              cpu: {{ .Values.resources.requests.cpu }}
```

Un file **_helpers.tpl** contiene implementazioni di funzioni personalizzate attraverso un blocco "define". La sua presenza nella cartella "/template" è opzionale cosi come il file "NOTES.txt", il quale è simile ad un README, stampato all'installazione del Chart per fornire una guida o una spiegazione generale.

4.2.3 Rilascio e installazione

Helm utilizza una CLI per interfacciarsi con l'utente e riceverne comandi. Una volta finita e completata la cartella del Chart, è possibile avviarne un'istanza attraverso il comando:

```
1    helm install <nome-release> <path/to/Chart.yaml/folder/>
```

E invece per fermare la release:

```
1    helm uninstall <nome-release>
```

Ovviamente il comando permette di settare valori aggiuntivi della release, come il namespace. Oppure è possibile passarvi un file "values.yaml" che sovrascrive quello presente nella cartella del Chart.
Essendo Helm un gestore di pacchetti, usando il comando "upgrade", è possibile passare da una Release che si basa su un Chart vecchio ad una nuova che si basa su una versione aggiornata (specificata in Chart.yaml). Il comando "rollback" permette di annullare immediatamente l'upgrade in caso di problemi ritornando ad una versione precedente.
Raggiunta una implementazione del Chart, templatizzata e parametrizzata correttamente, si è pronti per la pacchettizzazione ed il rilascio su un repository locale o in rete. Il comando "helm package ¡path/to/chart/folder¿/" crea un file .tgz con nome e versione specificate nei metadati all'interno di

Chart.yaml. Una volta pronto il pacchetto è possibile caricarlo in un repository come Helm Hub o come release su GitHub. Infatti Helm permette di importare repository di Chart esterni. Attraverso:

```
1    helm repo add myrepo https://example.com/charts/
```

E successivamente installarne un Chart che vi è all'interno con:

```
1    helm install releasename myrepo/mychart
```

A questo punto Helm svilupperà i manifest corretti e completi, se non vi sono errori di sintassi, e li passerà all'API Kubernetes che li schedulerà e li eseguirà.

Capitolo 5

PACKS, progettazione e architettura

Per risolvere i problemi sulla mancanza di risorse hardware e per semplificare il deployment di una serie di applicazioni containerizzate seguendo un approccio basato su servizi Container as a Service, nasce il progetto PACKS (Packaging Automated (for) Kuberneters Containers System). Questo progetto vuole, come già accennato in precedenza,fornire un sistema centralizzato per la gestione di container Docker in remoto. Ogni componente del progetto si trova a sua volta sullo stesso sistema Kubernetes con cui comunicherà. In particolare vi sarà:

- una **web app ExpressJS** a disposizione degli studenti, la quale semplificherà la macchinosa serie di comandi CLI richiesti per la semplice gestione dei container e permetterà la consegna del progetto personale al docente amministratore, facilitandone la presentazione e la correzione. Un login a server remoto LDAP permetterà l'accesso con credenziali universitarie;
- due **reverse proxy server** gestiranno il networking relativo ai percorsi della web app e i collegamenti diretti con i container, fornendo un livello basilare di autenticazione e privacy agli studenti utilizzatori;
- Un **sistema di caching e memoria persistente**, che fornisce autenticazione fra le varie componenti del progetto, ottenuto con **Redis**;
- una interfaccia scritta in GO, che grazie alle librerie ufficiali gestirà la comunicazione con Helm e Kubernetes.
- Un nuovo formato di pacchettizzazione verrà fornito grazie ad Helm e il suo template engine.

Nei prossimi paragrafi verranno analizzate la struttura e le scelte implementative fatte. Prima però bisogna parlare del sistema di base, Kubernetes.

5.1 Ambiente Kubernetes

Kubernetes è già stato ampiamente introdotto nei capitoli precedenti. Nel progetto PACKS costituisce la base di tutto. Esso è installato inizialmente su 2 nodi con 16 GigaByte di RAM l'uno e un decente numero di core CPU.

5.1.1 Installazione Control-Plane

Il Control-Plane richiede almeno 4 elementi applicativi da scaricare ed installare sulla macchina: kubeadm, kubelet, kubectl e containerd. Ogni nodo non deve avere una partizione swap abilitata, poichè non supportata da Kubernetes. Quindi prima di avviare qualsiasi installazione conviene disattivare temporaneamente lo swap con 'sudo swapoff -a' oppure, in maniera definitiva, modificando il file "fstab" e commentando la riga che indica il "mount" della partizione. Inoltre bisogna che i nodi abbiamo abilitato il forwarding dei pacchetti ipv4 e dei moduli kernel già installati come "overlay" e "br_netfilter". Il firewall di sistema potrebbe avere porte bloccate necessarie al funzionamento che bisogna aprire. A questo punto si potrebbe procedere all'installazione delle componenti.
Kubelet, come già visto, è il demone che si serve del Container Runtime Interface per avviare e gestire i container del nodo, al tempo stesso, interfacciandosi constantemente con il kube-api-server.
Kubeadm è il tool che permette di fare bootstrap dell'ambiente Kubernetes, in particolare avvia le componenti del Control-Plane e li configura in base ai parametri scelti dall'utente. In questo caso, dato l'uso di FLannel come CNI, viene passato come parametro la subnetwork dei Pod compatibile "10.244.0.0/16". Vengono avviati: etcd, kube-proxy, kube-api-server, kube-controller-manager, kube-scheduler e i coreDNS ed assegnati indirizzi Ip corrispondenti alla nuova configurazione.
A questo punto manca l'installazione del Container Network Interface per far comunicare tra loro i Pod. Verrà usato Flannel, il più semplice tra i CNI, nella sua configurazione standard con backend "Host-gw".
Al termine dell'esecuzione del comando di bootstrap 'kubeadm init' è possibile aggiungere nodi worker al cluster grazie al comando 'kubeadm join'. Questo significa che in ogni nodo è richiesto almeno kubelet, kubeadm ed il Container Runtime Interface.

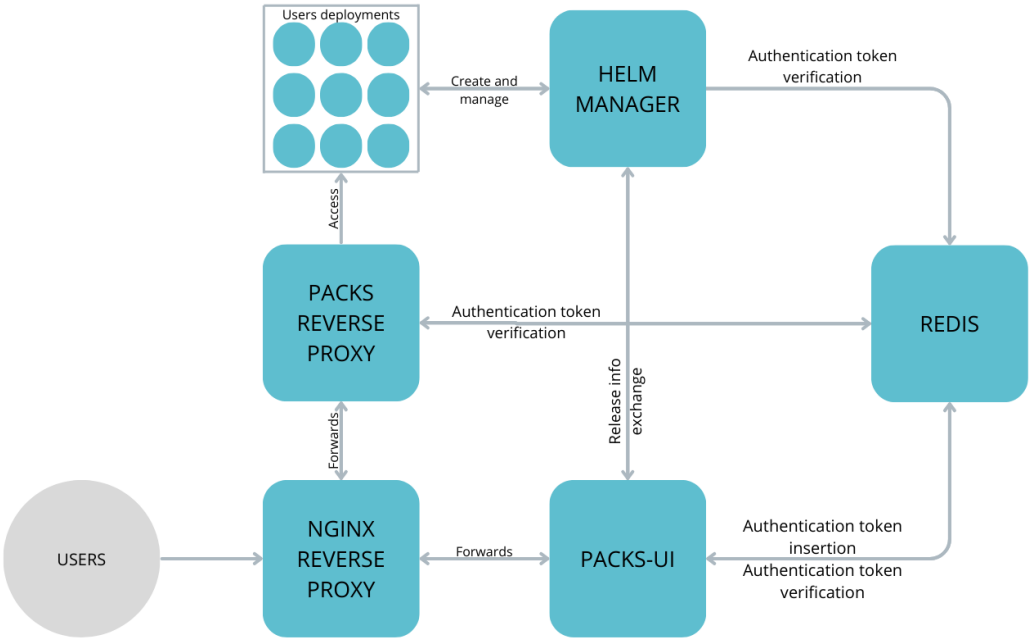

Figura 5.1: Pod del progetto PACKS

5.1.2 Installazione PACKS

Prima di tutto bisogna inserire il repository del Chart che contiene PACKS. Infatti è inserito come GitHub Page e si aggiunge con il comando

```
helm repo add <nome_del_repository> https://giuseppebnn.
    github.io/PACKS/
```

Una volta aggiunto il repository si può avviare il progetto con

```
helm install <nome_del_repository>/PACKS
```

Usando il tool "kubectl" si possono effettuare query di informazioni e comandi per applicare manifest. Con il comando

```
kubectl get pods
```

che effettuerà una query al kube-api-server restituendo un lista di Pod ed il loro stato. Da qui è possibile osservare le varie componenti: packs-ui, packs-reverse-proxy, helm-manager, Nginx-proxy e Redis.

5.1.3 Struttura generale

Lo schema 5.1 rappresenta i Pod di PACKS ed il loro modo di comunicare. Le componenti centrali, ovvero quelle che racchiudono la maggiorparte delle funzionalità, sono **Helm-Manager** e **PACKS-UI.** Mentre le altre, pur essendo più piccole, sono altrettanto importanti per garantire la corretta funzionalità del sistema.
L'utente parte collegandosi al dominio del proxy Nginx. Quest'ultimo, in base alla porta scelta indirizzerà a PACKS-UI, la webapp centrale dove lo studente farà login e gestirà i suoi progetti, oppure al reverse proxy interno. Tutte le query ed i comandi effettuati sull'interfaccia sono inoltrati e gestiti da Helm-Manager. Questa è l'interfaccia che gestisce a basso livello i template Helm e le risorse Kubernetes. Grazie a Redis e a il suo sistema di caching è possibile garantire un buon livello di autenticazione fra le richieste e le comunicazioni tra le componenti attraverso ad un sistema di Token JWT, approfondito più in avanti. IL reverse proxy di PACKS permette di inoltrare le richieste direttamente ai container degli utenti autenticati. Questi aspetti necessitano di essere approfonditi come si deve oltre alla brevissima introduzione ricevuta.

5.1.4 Nginx

Il server Nginx è la componente "esposta all'esterno" di PACKS. Esso è configurato come reverse proxy e filtra le richieste in entrata. Però, prima di parlare dell'implementazione, bisogna definire cos'è un "proxy" ed un "reverse proxy".
Un **proxy** è un server che fa da intermediario fra due macchine che comunicano in rete. Esso riceve le richieste da un client e le rispedisce in rete facendone le veci. Non appena riceve una risposta la inoltra al client. Quindi, in breve, è possibile dire che il proxy (o appunto il "forward proxy" in questo caso) nasconde il client alla rete esterna. I vantaggi che porta sono diversi. Oltre a garantire l'anonimato nascondendo l'indirizzo IP sorgente, fornisce la possibilità di aumentare le prestazioni grazie ad un sistema di caching per richieste multiple. Oltretutto, il proxy è usato per bypassare alcuni limiti e filtri di contenuti per aree geografiche, posizionandone una unità al di fuori dell'area ristretta.
Un **reverse proxy**, cosi come il proxy, si interpone tra un client e diversi server backend di un applicativo, inoltrandone le richieste a quello più appropriato e così nascondendone l'implementazione interna. Questo approccio permette di distribuire le richieste esterne a server specifici, decisi anche a runtime, garantendo non solo un bilanciamento del carico, ma anche un

miglioramento delle prestazioni, dato che molti step di autenticazione e decrittazione TLS/SSL possono avvenire nel proxy, alleggerendo l'elaborazione dati al backend.

Nginx, tra le sue funzioni, offre quella di server reverse proxy, permettendo il filtraggio fra le varie componenti di PACKS. La sua configurazione consiste nel instradare il traffico in due percorsi differenti: sulla porta 80 (nodePort 30080) verso la UI di PACKS e sulla porta 81 (nodePort 30081) verso il reverse proxy interno dei deployment utente.

Codice 5.1: nginx.conf di PACKS

```
1     events {
2       worker_connections 1024;
3     }
4     http {
5       client_max_body_size 10M;
6       server {
7         listen 80;
8         server_name packs.ui;
9         location / {
10            proxy_pass http://packs-ui-dp:80;
11        }
12      }
13      server {
14        listen 81;
15        server_name packs.proxy;
16        location / {
17            proxy_pass http://packs-proxy-dp:80;
18        }
19      }
20    }
```

In particolare, com'è possibile notare nel file di configurazione 5.1, oltre all'instradamento appena discusso, vi è impostato un limite nel body delle richieste per evitare un appesantimento della rete. Il file 5.1 è passato all'interno del Pod attraverso una ConfigMap e montato al percorso corretto attraverso Volumes e VolumeMounts.
Il Pod è gestito da un DaemonSet che ne replica una istanza in ogni nodo in modo tale da sfruttare le proprietà del Service associato ad esso. Ciò rende possibile una forma iniziale di load balancing leggero attraverso ad una distribuzione round-robin delle richieste. All'interno del file 'nginx-r-proxy.yaml' è visibile visionare il manifest completo della componente.

5.1.5 PACKS UI

Uno dei due "cuori" di PACKS, la User Interface offre una interfaccia web intuitiva e semplice con una grafica minimale e pulita. Come già introdotto in precedenza, essa centralizza le principali operazioni per la gestione dei container e ne aiuta a monitorare il comportamento attraverso la fruizione in unico luogo dei log. Inoltre permette un "forwarding" autenticato all'interno di ogni singolo container di un progetto, ove permesso.
Questa non è soltanto una interfaccia dedicata agli studenti ma anche al docente amministratore del corso, il quale può gestire la consegna e la valutazione del progetto tutto in un unico ambiente.
Nel dettaglio si tratta di una app sviluppata in NodeJS ed ExpressJS per il backend, in EJS e TailwindCSS per il frontend. La scelta di NodeJS ed ExpressJS è dovuta principalmente alla confidenza con il linguaggio di programmazione e le sue librerie, che permettono una gestione asincrona delle richieste e l'uso di una vasta serie di moduli "npm" (Node Package Manager).
Lo script consiste inizialmente in una serie di settaggi e definizioni di callback per ogni azione HTTP (GET, POST ...). Ad ognuna di queste funzioni viene effettuato il blocco di codice inerente la richiesta ed il risultato viene inserito in una variabile con notazione JSON e passata alla vista corretta di EJS.

5.1.5.1 EJS

Embedded JS è un motore di template di pagine HTML che permette un rendering dinamico con un linguaggio di programmazione simile a Javascript. I template, o "viste", si trovano nella sottocartella "views" della UI e permettono, in base alle variabili passate nelle callback, di effettuare il "rendering" dei dati al backend. Questo approccio offre una comoda gestione sul frontend evitando cosi appesantimenti nelle pagine dovuti a script per il fetching dei dati e conseguenti falle di sicurezza da curare.
Un esempio è la funzione "render", dell'app Express, che prende i input il nome della "view" template e i dati elaborati, passandoli a EJS che "renderizza" il tutto e lo manda in risposta al browser come pagina html. La vista è un file con estensione ".ejs" e consiste in tutto e per tutto in un file html con la possibilità di inserire condizioni e cicli simil-javascript in blocchi di codice delimititati da caratteri particolari.
Un esempio lo si trova nel codice 5.2, un estratto della view di login.

Codice 5.2: estratto di login.ejs

```
1 <% if(typeof isAdmin == 'undefined') {%>
2 <li>
3 <a
```

```
4 href="/upload"
5 class="text-secondary hover:text-secondary-focus text-sm md:
      text-xl"
6 >Upload</a
7 >
8 </li>
9 <% } %> <% if (typeof login == 'undefined') { %>
10 <li>
11 <a
12 href="/logout"
13 class="text-secondary hover:text-secondary-focus text-sm md:
      text-xl"
14 >Logout</a
15 >
16 </li>
17 <% }%>
```

Il modo d'uso e l'inserimento dei blocchi di codice è simile a quello di Helm e dei suoi templates. Questo fornisce un sistema modulare nella renderizzazione di pagine poichè nelle view si possono includere altre viste passandone di conseguenza altre insieme a nuove variabili. Ogni pagina della UI, infatti, include la view "header.ejs" oltre alla propria implementazione. L'uso di questi moduli è utile nell'aggiornamento e mantenimento di singole componenti.
Una strategia efficace, che fa buon uso delle proprietà di EJS, è quella che permette, invece di ricaricare la pagina dopo un certo di periodo di secondi, di richiedere solo la parte interessata e di sostituirla a quella vecchia. Nella dashboard dell'utente e nella pagina "details" viene usato questo stratagemma, aggiornando lo stato ogni pochi secondi oppure dopo ogni azione.

5.1.5.2 Autenticazione

L'autenticazione avviene in una pagina specifica, esattamente al percorso "/login". PACKS non richiede alcuna forma di registrazione perchè utilizza le credenziali uniche del portale studenti universitario. I parametri richiesti sono quindi il codice fiscale e la password del portale studenti. La verifica delle credenziali avviene attraverso una query al server LDAP dell'università, che racchiude i dati degli studenti.
Il **Lightweight Directory Access Protocol** è, appunto, un protocollo di accesso a dati e informazioni in rete presenti sotto forma di directory organizzate in gerarchie.
Ogni elemento memorizzato è identificato univocamente da un **Distinguished Name** (DN) che contiene una serie di elementi che ne formano la gerarchia nella directory. Fra essi vi sono:

- i **Domain Component**, attributi che indicano una parte di dominio di rete della risorsa;
- le **Organizational Unit** (OU) che distinguono gruppi o sottogruppi della gerarchia;
- i **Common Name** (CN) che rappresenta un nome, possibilmente ripetuto all'interno della directory, della risorsa in questione.

La struttura finale di un Distinguished Name sarà l'insieme di questi valori separati da virgola. Il seguente esempio indica il DN completo di una risorsa studente nel server LDAP della Università di Catania.

```
1    CN=<codice fiscale>,OU=Studenti,DC=unict,DC=ad
```

Il server può associare ad ogni risorsa una password. Attraverso il "bind" sulla risorsa è possibile effettuare un login a tutti gli effetti. In pratica il client invia il DN e la password, il server ne cerca la risorsa associata e verifica che la password fornita coincida con quella associata. Una risposta positiva è intesa come login riuscito, il contrario una negativa. Nella UI la componente che gestisce l'autenticazione LDAP è "ldapInterface.js" che a sua volta utilizza il package "ldapjs".
Una volta effettuata l'autenticazione, il mantenimento della sessione avviene attraverso l'utilizzo di Token JWT ed un relativo sistema di caching.
I JSON Web Tokens, o JWT, sono un formato standard per la creazione di stringhe cifrate adatte al passaggio di informazioni autenticate in rete, in particolare le informazioni di sessione. La struttura di un token, prima della cifratura, segue il formato JSON, infatti è possibile inserirvi qualsiasi tipo di informazione, chiamata "claim". Il contenuto, inoltre, comprende un segreto firmato atraverso un algoritmo di Hash, in genere SHA256. Il tutto viene cifrato in una stringa Base64URL che ne compatta il contenuto e lo rende più trasportabile via richieste HTTP.
Il ricevente può verificare l'integrità del token a la validità dei suoi claim verificando la firma. Se questa non risulta alterata allora è possibile estrarre i claim con la sicurezza che il mittente sia quello corretto. Ovviamente la sicurezza dei JWT cade quando il segreto viene scoperto, a quel punto un attaccante potrebbe tranquillamente replicarne i token con la firma corretta.
PACKS utilizza questi token per un sistema Single Sign-On (SSO), ossia la possibilità di accedere a tutti i servizi senza rifare il login ogni volta. Una volta che l'utente ha eseguito la POST su "/login", il backend esegue il bind LDAP al relativo server. Successivamente, se la risposta è positiva, viene creato un JWT con claims: il codice fiscale dell'utente, il ruolo "user", un timestamp di creazione e la firma. Il token viene dato in risposta, attraverso

il middleware "express-session", in un header "Authentication". In contemporanea al passaggio precedente, il JWT viene salvato in una cache Redis condivisa, con validità di qualche ora. Da quel momento in poi, fin quando non scade il token, l'utente può accedere a qualsiasi pagina all'interno del suo account poichè ad ogni richiesta annetterà l header "Authentication", che verrà letto e verificato ogni volta e per ogni azione.
Vi sono due ruoli all'interno della UI, quello di utente comune e quello di "administrator". Quest'ultimo è esclusiva del docente e viene assegnato nel claim "role" nel JWT. Molte azioni richiedono un token con ruolo di administrator, come la dashboard dei progetti consegnati (si vedrà più avanti).
L'ultimo ruolo presente nel sistema dei token è quello dei "deliver", ossia il token segreto, che solo un utente con ruolo amministratore può creare, per la consegna dei progetti studenteschi.

5.1.5.3 Dashboard Utente

Una volta autenticato, lo studente viene reindirizzato nella propria dashboard utente. In questa pagina è presente la lista dei progetti caricati, i quali possono essere avviati, fermati, rimossi e visionati nel dettaglio. La pagina dashboard ha uno script javascript che si occupa di riaggiornare ogni 10 secondi la lista delle release dei progetti ed il loro stato generale. In particolare lo script esegue una richiesta GET al percorso "/charts-status", il che innesca una funzione nel backend che inoltra una richiesta di informazioni, mantenendo il token, al Pod Helm-Manager.
In risposta si riceve lo stato, in formato JSON, che a sua volta viene passato ad una vista EJS, la quale, attraverso dei cicli, itera fra i progetti presenti e ne aggiunge i "button" corretti in base allo stato. Prosegue la sua funzione renderizzando il tutto ed inviando infine il risultato come risposta alla richiesta iniziale dello script, che si occuperà di sostituirla interamente alla lista già presente. Questo evita un fastidioso e invasivo ricaricamento della pagina intera ed un "più nascosto" e pulito aggiornamento del codice html.
In alto a destra, se autenticato, è possibile recarsi nella pagina di caricamento progetti.

5.1.5.4 Uploads

Nella percorso "/upload" si trova una pagina con un form che permette il caricamento di un nuovo progetto. Qui è possibile inserirvi il nome, il file .yaml del template PACKS e l'eventuale zip con i file da montare sui container. Per sicurezza è stato impostato un limite a 10 Megabyte del payload totale soprattutto per evitare congestionamenti ed appesantimenti involonta-

ri, o anche volontari, della rete. Più in avanti verrà trattata ed approfondita la sintassi del template PACKS.
Cliccato il pulsante "Upload" e spedito il "form" al backend, questo verrà inoltrato a Helm-Manager, il quale si occuperà di verificarne ed installarne il contenuto. Se la risposta è positiva allora si verrà reindirizzati nella dashboard dove apparirà la lista aggiornata con il nuovo nome e relative azioni, se invece vi è un errore verrà mostrato un messaggio apposito di allerta.
In un ambiente di produzione, tutte le comunicazioni attraverso le componenti, specialmente fra la UI e Helm-Manager, verrà fatta in sicurezza con HTTPS.

5.1.5.5 Dettagli release

Nella lista delle release nella dashboard vi è la possibilità, oltre a quella di avviare, fermare o cancellare i progetti caricati, di visionarne i dettagli e di consegnarli. Cliccando l'apposito pulsante, si viene indirizzati nella pagina "/details" del progetto. Qui è possibile, oltre a mantenere le stesse azioni della dashboard (tranne la cancellazione), visualizzare uno per uno i container che fanno parte della release insieme al loro stato, solo se il tutto è in fase "running".
In particolare vi è una griglia di elementi, ognuno contenente informazioni singole di stato, nome dei Pod, immagine usata e porte esposte. Il meccanismo di aggiornamento, allo stato dell'arte, è simile a quello usato nella dashboard, ossia il rinnovo di un singolo grande elemento della pagina piuttosto che tutto il documento. I dati aggiornati vengono poi ottenuti interfacciandosi con Helm-Manager e sempre usando le proprietà template di EJS.
Accanto ai pulsanti delle azioni ve n'è presente uno con la scritta "Deliver". Cliccandovi viene aperta una modale, con un form che prende in input un token avente come claim il ruolo "deliver".
Invece, cliccando il pulsante "Log" presente in ogni componente del progetto, è possibile visionarne direttamente lo standard output interno.

5.1.5.6 Logs

In particolare si viene reindirizzati al percorso "/logs/tokenIdentificativoProgetto/nomeDelPod" che effettua, attraverso uno script javascript interno e i parametri passati nella URL, una GET a "/dp-logs/tokenIdentificativoProgetto/nomeDelPod" che a sua volta effettua una query a Helm-Manager, richiedendo tutto lo standard output del Pod in questione. Questo contenuto viene poi assegnato alla view template "dp-logs-el.ejs" che ne controlla la presenza di un output e, se non presente, ne restituisce un eventuale messaggio di

errore.
Tali passaggi vengono effettuati ogni pochi secondi per tenere aggiornato l'output e per fornire una esperienza estremamente simile ad un comando "watch" su un "kubectl logs nome-pod" da terminale.

5.1.5.7 Connessione a porte esposte

Nella pagina dei dettagli, come già descritto prima, sono elencate per ogni singolo container le porte esposte, se presenti. Se una di questa, nel template PACKS, viene indicata anche come "hostPort", si potrà notare che, passandoci sopra col mouse, sarà cliccabile. Esattamente si verrà reindirizzati verso la possibile UI interna, esposta in un container, passando prima per il Reverse Proxy dei Deployment utente (trattato nel dettaglio più avanti).

5.1.5.8 Admin Dashboard

Una volta riconosciute le credenziali di administrator ed assegnato il token con claim il ruolo "admin", si viene reindirizzati nella dashboard apposita che differisce da quella dello studente per alcuni dettagli. Qui viene mostrata una lista di tutti i progetti consegnati con apposite azioni e dettagli. Così come l'utente, l'amministratore ha pieno controllo dei progetti consegnati, potendo accedere sia ai logs che alle porte interne.
Una differenza tra le dashboard la si trova nell'azione del pulsante di cancellazione, invece che cancellare e disinstallare il progetto da PACKS lo toglie solo dai progetti consegnati e quindi dalla specifica lista. La novità essenziale portata nella dashboard amministrativa è la sezione aggiunta sopra la lista delle release che permette, attraverso un click, di generare un token "deliver", necessario allo studente per la consegna.

5.1.5.9 Comunicazione con Helm Interface

Un occhio particolare va dato al modo in cui il backend NodeJS della UI comunica con Helm-Manager, la componente che gestisce le release degli utenti. Attraverso il package "helmInterface.js", creato appositamente per interfacciarsi con l'altra componente, le varie query sulle risorse Helm vengono gestite tutte in un unico file. Attraverso la libreria "axios", "helmInterface.js" effettua delle GET e delle POST inserendo correttamente nel body di ogni richiesta il token di autenticazione come "Authentication Header", fondamentale sia per mantenere un livello di autenticazione sia per riconoscere e trovare più facilmente le risorse associate a quel JWT.
Le risposte, che principalmente seguono un formato ben specifico, sono analizzate e pulite permettendo una gestione migliore in caso di malfunziona-

menti o contenuti indesiderati.
Principalmente il body delle richieste GET, che contiene solo il token, è usato per le query e la richiesta di informazioni come la lista delle release. Per le richieste specifiche, fra gli headers viene aggiunto anche il "referredChart" seguito dal valore del token JWT identificativo dei progetti. Quindi nelle richieste di dettagli per uno specifico progetto o per l'esecuzione di azioni come l'avvio, lo stop o la cancellazione, viene indicato anche questo parametro in più. Si vedrà nei paragrafi a venire il funzionamento della componente centrale per la gestione delle release PACKS.

Codice 5.3: esempio struttura richiesta axios in helmInterface

```
async function startChart(chartJwt, token) {
  try {
    const response = await axios.get(
      `${protocol}://${goServerIp}:${goServerPort}/install`,
      {
        headers: {
          Authorization: token,
          referredChart: chartJwt,
        },
      }
    );
    return true;
  } catch (error) {
    console.log(error);
    return false;
  }
}
```

5.1.5.10 Caching Redis

Una componente fondamentale in PACKS è Redis, un database "in-memory" che memorizza i dati sottoforma di elementi "chiave = valore". Memorizzando i valori in RAM, garantisce livelli di velocità altissimi oltre a supportare la scalabilità orizzontale su diversi nodi. Allo stato dell'arte attuale, un Pod, date le prestazioni che fornisce, basta e avanza. Il Pod communica e viene interrogato da quasi tutti gli elementi di PACKS, poichè è il posto dove vengono memorizzate informazioni chiave.
Il backend della UI, una volta che crea un token di autenticazione, attraverso il package "redisInterface.js" che fornisce funzioni specifiche per la comunicazione con Redis, crea un attributo "chiave=valore" che ha, come chiave, il token JWT utente e, come valore, il codice fiscale associato.
Successivamente tale attributo viene settato, in Redis, con una scadenza

uguale a quella impostata nell'header di sessione da "express-session". Questa scelta è stata fatta inizialmente per semplificare la gestione delle sessioni. Infatti il backend NodeJS, ogni volta che riceve una richiesta ne verifica la firma del token e ne controlla la presenza su Redis. Se presente allora è valido, poichè soltanto la UI inserisce questi valori di sessione. Inoltre la presenza dei JWT è fondamentale per il funzionamento di Helm-Manager.
I token "deliver", una volta creati dall'amministratore, sono poi anche "attivati", poichè vengono inseriti in una struttura dati importante di Redis, il SET. Questi permettono di associare più valori unici ad una chiave in particolare, cosa non possibile nelle classiche coppie "chiave=valore", dato che le chiavi devono essere sempre uniche. Infatti se si prova ad inserire due attributi con la stessa chiave, verrà preso in considerazione l'ultimo inserito, sovrascrivendone il valore precedente. I SET invece attribuiscono una lista ad un unica chiave.
I token vengono considerati attivi e validi quando vengono inseriti dentro questa lista. Una volta utilizzati in una consegna, vengono rimossi e quindi resi invalidi e non riutilizzabili.

5.1.5.11 TailwindCSS

Da come si può velocemente dedurre osservando le views EJS, lo stile delle componenti grafiche della UI è affidato alle classi TailwindCSS, un framework CSS considerato "utility-first", ovvero che non richiede codice CSS specifico per le componenti HTML ma reisce a creare qualsiasi stile attraverso l'esclusivo uso delle classi e la loro combinazione. Grazie ad esso la UI ha un design "responsive" e cambia adattandosi alle dimensioni dello schermo, che sia esso di uno smartphone o di un computer.
La scelta è ricaduta su TailwindCSS anche grazie alla sua totale compatibilità con EmbeddedJS oltre che alla sua semplicità.

5.1.5.12 Immagine e Manifest

Una volta definite le principali funzioni e spiegati i passaggi più importanti che compie PACKS UI, bisogna trattare ed approfondire come l'intera componente venga trasformata in un immagine Docker.
Attraverso il Dockerfile nel codice 5.4 è possibile vedere come si parta da un immagine di base con NodeJS e NPM preinstallati, vi si copino tutti i file necessari e di come vi si installino le dipendenze necessarie. La porta esposta del container è la 3000 e l'Entrypoint è subito settato per eseguire lo script principale.

Codice 5.4: Dockerfile PACKS UI

```
1 FROM node:latest
2 COPY . /packs-ui
3 WORKDIR /packs-ui
4 RUN npm install
5 RUN npm run build
6 EXPOSE 3000
7 ENTRYPOINT ["/bin/bash", "-c", "npm run start"]
```

Attraverso un sistema di GitHub workflows, questa immagine viene ricreata e inserita in un repository DockerHub.
In Kubernetes, PACKS UI viene rilasciata nel cluster come Deployment, attualmente ad una replica. Il container all'interno del Pod prende l'immagine aggiornata direttamente dal registry DockerHub, ne apre la porta 3000 e vi monta il file delle variabili ".env". Non è stato accennato prima ma è stato prediletto l'inserimento di variabili sensibili e di configurazione attraverso il package "dotenv". Nel Chart Helm finale sarà necessario un file values che contenga le variabili .env come di seguito:

Codice 5.5: esempio variabili di ambiente nel values.yaml

```
1 env:
2   REDIS_H: redis
3   REDIS_P: 6379
4   PROXY_URL: http://proxyurl.com
5   JWT_SECRET: a very big secret
6   SESSION_SECRET: important secret to change
7   PROXY_SECRET: a big secret that has to change and has to be
        the same on both backend and proxy
8   JWT_ADMIN_SECRET: another big secret for the admin user
9   JWT_DELIVER_SECRET: another big secret for delivering
        deployments
10  LDAP_URL1: url/server/ldap1
11  LDAP_URL2: url/server/ldap2
```

Helm, attraverso il manifest "env-cfmap.yml" presente tra i template del Chart finale, genererà in automatico la ConfigMap iterando nel file "values.yaml". Quelle elencate, in particolare, sono informazioni che indicano come raggiungere componenti interne del Chart di PACKS, come Redis e il proxy dei deployment. Per tal motivo alcune di queste saranno lasciate, in futuro, "hardcoded" nel Chart finale e quindi non più necessarie nell'elenco.
Un service con hostname "packs-ui-dp" completa la componente fornendo una interfaccia di rete statica all'interno del cluster, esponendo la porta 80 e collegandola alla 3000 della UI.

5.1.6 Helm-Manager

Come il nome suggerisce, questa componente fondamentale di PACKS è l'interfaccia principale per tutto quello che riguarda la gestione diretta dei progetti su Helm e Kubernetes.
Scritto in Go, si espone sulla porta 9000 ed ascolta le richieste che arrivano da PACKS UI, gestendole attraverso una serie di middleware creati appositamente. Vari package si suddividono gli ambiti nella gestione del sistema sottostante e sono:

- **main**, si mette in ascolto sulla porta 9000, gestisce le varie richieste in base all'URL, compone e assegna i middleware per la loro gestione e verifica i token ricevuti;
- **httpHandler**, package principale che racchiude in se le implementazioni di tutti i middleware che, a loro volta, implementano il codice necessario per la corretta comunicazione con la UI. Esegue le corrette funzioni degli altri package e ne gestisce l'esito, inglobandolo in un formato di messaggio standardizzato e restituendolo in risposta alla UI;
- **relHandler**, è il package più grande del lotto, si occupa dell'installazione dei progetti, della loro verifica e tutto quello che li concerne. Utilizza "helmInterface" per l'installazione del template PACKS e "K8sInterface" per effettuare query direttamente a Kubernetes. Inoltre si interfaccia costantemente con Redis attraverso "redisInterface" per la gestione delle informazioni sui progetti;
- **helmInterface**, package che gestisce direttamente il client Helm, utilizzandone le librerie ufficiali (helm.sh/helm/v3);
- **K8sInterface**, gestisce e consente la comunicazione con le API di Kubernetes, sfruttandone le librerie ufficiali (k8s.io);
- **redisInterface**, attraverso package specifici, permette di comunicare con il client del Pod Redis;

Data la complessità generale di Helm-Manager, per avere un'idea più chiara sulla sua implementazione, conviene analizzare una per una le funzionalità principali, partendo dal "main" package.

5.1.6.1 Richieste e middleware

Tutte le richieste gestite dal file "main.go" sulla porta 9000 passano attraverso una pipeline formata da middleware. Questa viene assemblata all'avvio

del server e viene affidata ad una funzione "Handle" del package "http".
Ne viene assemblata una per ogni richiesta, poichè queste necessitano di essere gestire diversamente. Nel dettaglio, i primi due Handler della pipeline sono sempre uguali, quello che differisce è sempre l'ultimo elemento, ossia l'esecutore vero e proprio della richiesta. I primi due si occupano di verificare la bontà e la validità della richiesta, controllando l'origine attraverso il token e che le regole CORS siano rispettate.
Il terzo Handler, come si puo vedere nel codice del file "main.go", gestisce la richiesta ed esegue la serie di funzioni utilizzando gli altri package a disposizione. La pipeline viene composta attraverso una funzione implementata nel file "httpHandler.go", il quale prende in input gli Handler ed in ordine li collega in un solo middleware.
Detto questo, per parlare di implementazione vera e propria del sistema, bisogna approfondire gli ultimi Handler della catena.

5.1.6.2 Upload

Quando l'utente clicca il pulsante "Upload" nella UI, attraverso il backend, la richiesta viene inoltrata, con il template PACKS all'interno del body, verso Helm-Manager. Nello specifico, il "main.go" gestisce questa POST attraverso una pipeline che ha come Handler finale "Upload Handler" di "httpHandler.go".
Qui avvengono i primi controlli: viene verificato innanzitutto il numero di progetti già caricati dall'utente associato al token di Autenticazione e se questo corrisponde ad un numero minore di 2, allora nessun problema, tale limite è stato scelto per tenere una quantità controllata di progetti. Per fare ciò, viene chiamata una funzione di "relHandler.go" che a sua volta utilizza redisInterface.go per fare le query e calcolarne il numero. Come si vedrà, ogni progetto caricato viene associato ad un utente e memorizzato in Redis attraverso l'uso di SET. Se il sistema vede che vi sono già presenti due progetti allora viene annullato l'upload, ritornando un messaggio di errore indietro fino alla UI.
Se il metodo è POST ed il numero di progetti è massimo 1, si procede con la creazione di un nuovo nome in codice del progetto, in particolare un JWT abbreviato a circa 50 caratteri per rispettare le limitazioni Kubernetes. Grazie alle funzionalità di "relhandler.go", viene creata una cartella con nome il token appena creato e si procede con l'estrazione del .zip dei file da montare, se presente nel body della richiesta, e successivamente con la copia del template PACKS nella nuova directory.
Questa installazione avviene all'interno di una cartella "/shared/" condivisa fra i nodi, in particolare nel percorso "/shared/uploads/". Le cartelle dei

progetti vengono create qui dentro e, trattandosi di una cartella condivisa, possono accedervi i container del progetto da ogni nodo che la monti nel suo file system. I due server su cui è installato PACKS condividono tale directory e quindi qualsiasi nodo è in grado di montare sui container i volumi e/o i file richiesti per il funzionamento.
A questo punto l'interno della directory di un progetto installato si presenta cosi:

```
/m1njq3ntf9bzht2fwo5ly5gkcqwhgjs40vbzxaxjv4f2puen1m
  -  /mnt/
      | - /directoryDaMontare1/
      | - fileDaMontare1.py
  -  values.yaml
```

Una volta finita l'installazione fisica si procede a registrare l'avvenuto caricamento su Redis attraverso la "relHandler.SaveToRedis()" che, attraverso redisInterface.go, memorizza in un SET con chiave il codice fiscale dello studente, una stringa json di informazioni come nome, token identificativo del progetto e namespace. Quest'ultimo è estratto da un altro token JWT facendo in modo che sia unico e difficilmente uguale ad altri, si vedrà più in avanti il perchè.
Se tutti i passaggi sono andati bene e tutte le funzioni non hanno dato errori allora httpHandler fornirà una risposta HTTP con status 200.

5.1.6.3 List

Non appena viene ricevuta una richiesta "/list", il middleware incaricato di gestirla ha come ultimo Handler quello fornito dalla funzione "ListHandler" di httpHandler.go.
Se la richiesta corrisponde ad una GET, la palla passa al package relHandler che, una volta interrogato Redis sui progetti collegati al token ricevuto, fa una query al client Helm attraverso il package helmInterface per vedere quali di queste sono attive e quali no. Ottenuti i progetti ed il loro stato, le informazioni vengono inserite dentro un Message (messaggio in formato standard) e inviate in risposta alla richiesta GET.
Come visibile dal codice, helmInterface si basa su strutture, funzioni e configurazioni della libreria ufficiale di Helm e Kubernetes. In particolare quando si deve eseguire un comando su Helm, viene innanzitutto creato un oggetto 'action', della libreria "helm.sh/helm/v3/pkg/action". In questo caso l'oggetto interessato è "NewList". Una variabile di tipo 'action.Configuration' che racchiude la configurazione del client Helm e dell'ambiente Kubernetes (kube_config). Passata in input la configurazione al factory method NewList, si ottiene l'oggetto List, che rappresenta in tutto e per tutto il comando

da CLI "helm ls". A questo punto basta solo eseguire il metodo Run() dell'istanza List e riceverne l'output, il quale sarebbe un'array di "Realease". Questi oggetti contengono al loro interno le informazioni basilari delle release Helm avviate. Da qui vi si estrae lo stato e lo si abbina alla lista dei progetti recuperata da Redis.

5.1.6.4 Install

L'Handler finale della pipeline per questa richiesta è "httpHandler.InstallHandler". Con "install" non si intende una nuova installazione poichè quella è già avvenuta nell'upload del progetto, ma si intende il comando della CLI di Helm che avvia una nuova release nell'ambiente Kubernetes sottostante.
La richiesta accettata è una GET su percorso "/install" ed una volta estratti i due token dal body (uno utente e uno identificativo del progetto), InstallHandler si affida a relHandler eseguendo la funzione "InstallRelease". Questo metodo esegue prima di tutto una serie di verifiche che coinvolgono interrogazioni a Redis ed a helmInterface. In particolare si verifica l'esistenza del progetto fra le installazioni e se questo non sia già stato avviato. Una volta constatata la regolarità e la fattibilità della richiesta, si procede a creare un oggetto Chart di Helm attaverso helmInterface, configurato con il template PACKS (approfondito più in avanti) ed il token identificativo del progetto. Si procede con il caricamento del file values.yaml ricevuto in upload dallo studente e la creazione di un namespace dedicato. La funzione Install di helmInterface è pronta a ricevere questi parametri ed a chiamare il factory method per la "action.Install" di Helm e successivamente il metodo Run() dell'istanza appena creata.
Helm passerà tutti i manifest renderizzati al kube-api-server che li schedulerà nel miglior modo possibile. La risposta alla richiesta GET sarà ovviamente positiva se tutte le verifiche e controlli effettuati non daranno errori o problemi, in tal caso sarà inviata una risposta con codice negativo (5xx).

5.1.6.5 Delete

La richiesta GET "/delete" contiene nel body i due token JWT, come tutte le richieste per azioni su progetti specifici. I valori vengono estratti da "httpHandler.DeleteHandler" e passati a "relHandler.DeleteRelease". In questo caso i controlli preventivi sono diversi, in particolare vengono verificate:

- la presenza del progetto fra quelli installati
- l'appartenenza del progetto da cancellare al token utente inviato, se quest'ultimo non corrisponde al proprietario viene restituito un errore;

- lo stato del deployment, se attivo non permette la cancellazione;

A questo punto vengono rimossi tutti i dettagli del progetto dal SET Redis attraverso "redisInterface.DeleteFromSet", viene eliminata la directory di installazione in "/shared/uploads/" e viene cancellato il namespace unico da Kubernetes attraverso "k8sInterface.RemoveNamespaceIfExists". Per non essere più visibile e utilizzabile al frontend dall'utente basterebbe la sola rimozione da Redis, ma ovviamente per una questione di pulizia viene ripulito anche lo spazio su disco.

5.1.6.6 Stop

La richiesta GET "/stop", gestita da "httpHandler.StopHandler", fa parte delle richieste specifiche. Una volta estratti i token si esegue la "relHandler.StopRelease". Gli unici controlli che vengono effettuati permettono l'esecuzione dello stop, quindi la rimozione della instanza attiva in Helm, solamente una volta verificato l'esistenza di quest'ultima e che sia avviata. Un release non attiva non può essere fermata e rimossa con "helm uninstall" d'altronde. La corretta esecuzione ritorna una risposta con stato 200.

5.1.6.7 Details

Come la "/stop" e la "install", la GET "/details" contiene i token utente e progetto. Una volta estratti nella "httpHandler.DetailsHandler" vengono passati alla "relHandler.GetReleaseDetails", la quale, dopo aver verificato al solito l'esistenza con redisInterface e lo stato del progetto attraverso helmInterface, si affida al metodo "GetDeploymentsDetails" del package "k8sInterface". In questa funzione vengono utilizzate le librerie di Kubernetes per la creazione di un "clientset" configurato con il corretto file "kube_config" e per la comunicazione, attraverso le interfacce che fornisce con i metodi, con l'API di Kubernetes.

Codice 5.6: estratto da k8sInterface.GetDeploymentFromNamespace

```
1 pods, err := clientset.CoreV1().Pods(namespace).List(context.
    Background(), metav1.ListOptions{LabelSelector: "app=" +
    deploymentName,})
```

Il breve estratto del codice appena scritto rende l'idea delle interfacce che mette a disposizione la libreria Kubernetes. L'istanza "clientset" di "kubernetes.ClientSet" espone un metodo che restituisce un'interfaccia, in questo caso il "CoreV1" che, a sua volta, attraverso un factory method "Pods", restituisce un altra interfaccia per il namespace indicato. Successivamente la "List" effettua la query a Kubernetes restituendo un oggetto "v1.PodList",

il quale contiene appunto una lista dei Pod all'interno e tutte le informazioni, come porte, immagine, volumi ecc... Al metodo basta solo la presenza del namespace perchè ogni progetto ha il suo, univoco, quindi non si rischia di accedere a informazioni di altri Pod di altri utenti.
Una volta terminato il metodo di "k8sInteface", "relHandler" prende le informazioni basilari da Helm e vi aggiunge quelle appena ottenute da Kubernetes, spedendo il tutto in un Message in formato json.

5.1.6.8 Logs

Come le richieste specifiche precedenti, la "/logs" viene gestita da "httpHandler.LogsHandler" che, oltre ad estrarre i soliti 2 token, estrae anche il nome del Pod interessato e passa i valori a "relHandler.GetReleaseLogs". Quindi questa è una richiesta ancora più specifica poichè i dettagli voluti riguardano il singolo container all'interno di un progetto. A questo punto, oltre ai soliti controlli preventivi, vengono effettuate delle query sullo stato del progetto e sull'output del container del Pod indicato.
In particolare, questa ultima funzione viene eseguita in "k8sInterface.GetLogsFromPods" che, a sua volta, utilizza le ormai già citate interfacce della libreria Kubernetes. Attraverso il factory "CoreV1().Pods(namespace)" restituisce una interfaccia "PodInterface" per il namespace indicato. Questa contiene il metodo "GetLogs" che restituisce una richiesta REST preconfigurata per l'API Kubernetes. Uno stream di informazione viene restituito dal metodo "Stream()" dell'istanza "rest.Request" appena creata e ne viene memorizzato il contenuto in un buffer. Questo viene ritornato a relHandler che elabora i valori accumulati e li incapsula in un json, successivamente mandato in risposta alla GET.

5.1.6.9 Delivered e Undeliver

Le ultime due richieste gestite in Helm-Manager sono le GET "/delivered" e "/undeliver". Esse si occupano rispettivante della consegna e della rimozione del progetto dal gruppo una volta corretto dall'amministratore. Come già accennato in precedenza, il sistema di consegna si basa su dei token particolari con ruolo "deliver" creati a attivati soltanto dall'amministratore ed usabili una singola volta dall'utente. In Redis vi sono due SET funzionali alla memorizzazione ed alla consegna di tali progetti. Uno, come già visto, è il SET dei token "deliver" validi e non ancora usati, l'altro è un SET che contiene tutte le info dei progetti consegnati seguendo lo stesso schema dei SET per i progetti dell'utente. Praticamente, alla consegna, viene preso il record del progetto dall'utente e "copiato" nel SET admin. Quindi, a livel-

lo concettuale, il deployment acquisisce un nuovo proprietario, il quale può accedervi con gli stessi "poteri". Quindi la GET "/delivered", gestita da "httpHandler.DeliveredHandler", estrae i 2 token classici e li passa a "relHandler.DeliverRelease". A sua volta, come descritto in precedenza, copia i dettagli da un SET utente a quello admin. Effettuando questo semplice passaggio la consegna è completata.
La dashboard amministrativa della UI, quando fa la query dei progetti consegnati, effettua gli stessi passaggi per la dashboard utente classica, semplificando cosi il codice e risparmiando ulteriori complicazioni.
Invece la richiesta GET "/undeliver" viene presa in carico da "httpHandler.UndeliverHandler", il quale chiama la "relHandler.UndeliverRelease" passandovi i token estratti. A questo punto i passaggi sono semplici, dopo aver verificato l'esistenza del progetto lo si toglie dal SET admin. Quest'ultimo non risulterà più come un proprietario e il progetto rimarrà all'utente, il quale ha il vero potere di eliminazione totale dal sistema.

5.1.6.10 Containerizzazione e Deployment K8s

Il Dockerfile presente nella directory di Helm-Manager si occupa di fare il "build" dell'immagine container della componente. In particolare, la base è formata da una immagine ufficiale Golang su cui vengono copiati i file ".go", insieme al "go.mod" ed il "go.sum". Un layer fondamentale è quello che esegue il comando "go build -o helm-manager", ovvero la creazione di un file eseguibile. Infatti, dopo aver esposto la porta 9000, l'Entrypoint sarà settato sull'esecuzione diretta dell'eseguibile.
Il vero e proprio build dell'immagine, cosi come PACKS UI, viene effettuato tramite GitHub Workflows ad ogni aggiornamento del codice sorgente nel repository, infatti grazie alle "actions", l'immagine neogenerata viene caricata sul registry DockerHub.
Il rilascio sul cluster Kubernetes avviene sotto forma di oggetto Deployment. Innanzitutto, allo stato dell'arte, è necessaria la creazione di un Persistem Volume ed un Persistent Volume Claim per il funzionamento. Questi sono particolarmente rilevanti soprattutto per il montaggio della cartella condivisa "/shared/uploads/" ove si trovano le installazioni dei progetti ed i file da montare nei container che li compongono. Allo stato attuale il PV ed il PVC garantiscono permessi di lettura e scrittura multipli in uno storage di 2 Gigabyte. In futuro è pressocché sicura una scalata verticale di tali risorse data l'iniziale quantità.
Un altro elemento essenziale è il caricamento di una ConfigMap con all'inter-

no il "kube_config" dell'ambiente sottostante per far si che le librerie ufficiali di Helm e K8s possano comunicare con l'API server all'esterno.
Sostanzialmente, il container descritto nel manifest utilizza l'ultima versione presente nel registry DockerHub (giuseppebonanno99/helmmanager:latest) appena discussa, espone la porta 9000, monta il Persistent Volume Claim in "/shared/uploads" e la ConfigMap kube_config sotto forma di file. Una variabile d'ambiente "KUBECONFIG" indica a Helm-Manager dove trovare la configurazione caricata.
L'ultimo elemento del manifest per questa componente fondamentale di PACKS è il Service, il quale fornisce una interfaccia di rete statica all'interno del cluster, inoltrando su porta 9000.

5.1.7 Reverse Proxy interno

Si è visto come in PACKS UI le informazioni vengano prese attraverso Helm-Manager e mostrate a schermo nei vari dettagli, in particolare nella sezione "/details" di un progetto dove sono presenti ed elencate anche le porte esposte dai vari container. Se una di queste è di tipo "nodePort" sarà cliccabile ed accedibile via browser dall'esterno. Questo è permesso grazie alla componente **PACKS-proxy**, la quale, attraverso l'implementazione di un reverse proxy, indirizza le varie richieste verso i container interni dei vari progetti.

5.1.7.1 Preparazione alla connessione

Il tutto parte dall'utente che clicca sulla porta nodePort di un container. Questo esegue una richiesta GET al percorso "/forward-to-port/:chartJwt/:service/:port/:namespace" ricevuta e gestita dal backend della UI, dove, dopo aver controllato ed autenticato il token, vengono estratti i valori passati nel 'path' della GET. A questo punto avviene la creazione e la firma di un nuovo tipo di token specifico che ha come 'claims' i valori estratti in precedenza. Il JWT viene memorizzato attraverso "RedisInterface.js" come chiave di una stringa in formato JSON dei valori nella richiesta. Un nuovo URL viene costruito assemblando insieme l'indirizzo del PACKS-proxy, il percorso "connecttopodinport" ed il token appena generato.

```
1  const newUrl = `${proxyUrl}/connecttopodinport/${newJwt}`;
2  res.redirect(newUrl);
```

Come dal codice appena mostrato, l'utente viene reindirizzato all'indirizzo e percorso appena creato.

5.1.7.2 Implementazione e connessione

La connessione avviene attraverso una prima richiesta fondamentale verso l'URL generato in precedenza, in modo tale che il reverse proxy possa prima di tutto generare un nuovo instradamento e memorizzarlo.
Scendendo nel dettaglio, nella richiesta iniziale vi è presente un JWT fra i parametri passati, il quale subirà un controllo di esistenza attraverso "redisInterface.js". Dopo questa serie di passaggi, reindirizza l'utente verso il percorso nullo ("/"). Da questo momento in poi qualsiasi richiesta verso Packs-proxy sarà gestita e indirizzata in base ad un modello specifico. Il reverse proxy riceve la richiesta di un "path" variabile qualsiasi, ricevendo una stringa in json ed estraendo i valori come service, namespace e porta, prosegue con la combinazione di questi nel percorso di destinazione all'interno del cluster.

```
1 `http://${values.service}.${values.namespace}.svc.cluster.
    local:${values.port}`;
```

Com'è possibile notare dalla riga estratta sopra, il dominio per comunicare con un Service fra namespace diversi segue un formato specifico. A questo punto basterebbe inoltrare le richieste al nuovo URL creando l'istanza proxy per questa destinazione. Ma questo sistema deve ricevere un gran numero di richieste da inoltrare e creare sempre una nuova istanza per anche percorsi uguali è estremamente controproducente. Usando una variabile Map di NodeJS, attraverso la memorizzazione con attributi chiave/valore si riesce a memorizzare il proxy per un percorso specifico riutilizzandolo e non ricreandolo ogni volta, ad ogni richiesta. In tal caso la chiave è l'URL target e il valore l'istanza già creata. Per non affollare con una miriade di istanze proxy la memoria vi si assegna un tempo di vita massimo in modo tale che, scaduto il periodo, la memoria venga liberata di quell'elemento non più utilizzato.
Per implementare questo reverse proxy sono state usati ExpressJS ed "express-http-proxy" oltre al già descritto "redisInterface.js". Questo elemento, allo stato dell'arte, è ancora alle prime fasi di sviluppo e quindi contiene diversi potenziali difetti da correggere e funzionalità da migliorare, però la prima versione descritta in questo testo permette l'accesso senza problemi alla maggiorparte delle UI nei vari container dei progetti. Il sistema richiede un livello di autenticazione maggiore dato che c'è uno scarso controllo sul token utente e senza dubbio un sistema funzionante di comunicazione in HTTPS invece che in semplice e poco sicuro HTTP, come il resto delle componenti d'altronde.

5.1.7.3 Containerizzazione e Deployment

L'immagine definita nel Dockerfile, presente nella directory di PACKS-proxy, è piuttosto semplice e non contiene molti layer. Le fasi principali della sua costruzione sono la copia dei file sorgente, l'installazione delle dipendenze con "npm install", l'esposizione della porta 3001 su cui ascolta il proxy ed il settaggio dell'Entrypoint con l'avvio dello script principale. Il tutto ovviamente basato su un'immagine NodeJs ufficiale e caricato su un registry DockerHub, attraverso un workflow di GitHub, ad ogni modifica al codice.
L'utente, cosi come per la UI, deve passare per Nginx Reverse Proxy, porta 81, se deve raggiungere PACKS-proxy, poiché questa componente è rilasciata sotto forma di oggetto Deployment nel cluster Kubernetes e quindi non raggiungibile dall'esterno con un semplice Service ClusterIP.
Il manifest descrive un semplice Deployment Kubernetes con immagine l'ultima versione caricata nel registry e la porta 3001 esposta. Il Service che interfaccia la componente è molto semplice e si limita a inoltrare le richieste ricevute sulla propria porta 80 alla 3000 del PACKS-proxy.

5.1.8 Redis

Fino ad ora questa componente è stata affrontata molteplici volte nell'arco dei vari capitoli. Concentrare le sue funzionalità, dedicandole un paragrafo a sé, è pressocché impossibile data l'importanza del suo ruolo nelle varie funzionalità in PACKS.
In un unico luogo è presente sia un sistema di caching per token di sessione che uno storage persistente di informazioni importati e progetti. Questo fa comprendere l'enorme potenzialità di un software come Redis ed i suoi potenziali usi. Come si è già visto, si tratta di un database collocato in memoria centrale, super responsivo e scalabile, memorizza in formato "chiave/valore" le informazioni.
Vi sono diverse strutture, ognuna con le sue proprietà di memorizzazione, che vengono incontro allo sviluppatore offrendo diverse possibilità. Quello incontrato diverse volte all'interno di questo progetto, a parte la semplice tupla "chiave=valore", è il SET. Attraverso questa struttura sono state implementate, in maniera piuttosto semplice e minimale, funzionalità che in altri sistemi sarebbero potute divenire molto più complesse come la consegna o l'inserimento di progetti. Ma ancora le possibilità d'utilizzo per feature future sono numerose e quindi si potrebbe affermare che, usando Redis, si è fatta la scelta più azzeccata.
Una parte poco approfondita di questa componente è senza dubbio il modo in cui è rilasciata sul cluster Kubernetes. Redis ha un sistema di sincronizzazio-

ne con la memoria persistente sottostante per evitarne sconvenienti perdite a seguito di crash o malfuzionamenti della centrale. Il manifest, infatti, definisce un Persistent Volume ed un Persistent Volume Claim sul percorso "/shared/pv-redis/", i quali vengono montati nel container, con immagine "redis:latest", nel percorso "/data". Ad ogni riavvio il "dump.rdb" memorizzato nel file system del nodo viene letto e lo stato della memoria centrale ripristinato, permettendo sin da subito l'uso del servizio.
Il Service associato al Deployment è un classico e semplice ClusterIP con porta 6379, ossia la default Redis.

Capitolo 6

PACKS package

I capitoli precedenti hanno descritto sufficientemente gli aspetti e le funzionalità di PACKS, dei sistemi su cui si basa e le componenti che lo formano. Partendo dai reverse proxy, passando per la UI, fino ad arrivare ad Helm-Manager, le caratteristiche sostanziali sono state trattate tutte eccetto una, la più importante per lo studente che vuole utilizzare questo sistema, ossia la creazione e la pacchettizazzione di una serie di software containerizzati secondo il template PACKS.

6.1 Composizione

Principalmente un **package PACKS** è formato da 2 elementi: un file con estensione "yaml" ed un file zip. Il primo è un file di configurazione dichiarativo ove è possibile elencare tutti i container e la loro configurazione in modo semplice ed intuitivo. La sintassi trae ispirazione da quella di Docker Compose.

6.1.1 Docker Compose

Generalmente, per creare ed avviare un insieme di container Docker, l'utilizzo di diverse istruzioni in linea di comando era fondamentale. In molti casi, gestire tale molteplicità diveniva abbastanza macchinoso e problematico. Con l'introduzione di Docker Compose si riuscì a centralizzare ed a raccogliere in un unico file di configurazione dichiarativa Yaml, lo stato finale desiderato di qualsiasi container. Ogni parametro di configurazione non doveva più essere passato in linea di comando ed ogni container non necessitava più una gestione a sé. Inoltre la sintassi offerta offriva una semplicità nella lettura e comprensione del sistema di applicazioni containerizzate offrendone una im-

portante visione d'insieme.
La descrizione dichiarativa è rappresentata nel file "docker-compose.yaml" di seguito:

Codice 6.1: esempio di docker-compose.yml

```
1 version: '3'
2 services:
3   web:
4     image: nginx
5     ports:
6       - "8080:80"
7     volumes:
8       - ./html:/usr/share/nginx/html
9     healthcheck:
10      test: ["CMD", "curl", "-f", "http://localhost:8080"]
11      interval: 20s
12      timeout: 10s
13      retries: 10
14  db:
15    image: postgres
16    environment:
17      POSTGRES_PASSWORD: example
18    volumes:
19      - db_data:/var/lib/postgresql/data
```

In un solo file è possibile indicare qualsiasi tipo di configurazione e azione permessa dalla Docker CLI. I container, elencati per nome nella sezione "services", hanno ognuno, sotto forma di attributi "chiave:valore", la possibilità di indicare:

- l'immagine di base, ricercabile nel registry DockerHub o da costruire attraverso un Dockerfile;
- le porte da esporre e dove montarle, ad esempio nel 6.1, il servizio "nginx" è raggiungibile dall'esterno sulla porta 8080 del calcolatore sottostante;
- i volumi da montare, specificando il percorso di origine e di destinazione;
- le variabili d'ambiente valide all'interno del container;
- gli Healthcheck, similmente a quelli Kubernetes, forniscono istruzioni sul come e quando effettuare le verifiche di stato;
- i comandi da eseguire attraverso "command";

- un Entrypoint, a tutti gli effetti come quello di un Dockerfile;
- la possibilità di avvio del servizio in un certo ordine, aspettando che l'Healthcheck di qualche altro container sia positivo;
- la creazione e l'uso di reti virtuali interne, definite al di fuori dalla sezione "services";

Come si nota fortemente dalle funzionalità appena citate, un file Compose potrebbe tranquillamente sostituire qualsiasi Dockerfile poichè permette di fare molto di più in un unico luogo.
Docker Compose è uno strumento estremamente utile e potente per chi si approccia nel mondo delle applicazioni containerizzate, girando però necessariamente su una singola macchina (quella host), i problemi dell'utilizzo spropositato e di consumo delle risorse permangono invariati. Poi è pressocché impossibile rilasciare in produzione un sistema attraverso un file "compose.yaml" poichè non garantisce alcuna proprietà di High Availability, quindi tantomeno lo scaling e la fault tolerance. Il suo utilizzo ideale permane in un ambiente puramente di sviluppo e/o di testing personale per il rilascio successivo in ambienti di produzione distribuiti come un cluster Kubernetes.

6.1.2 values.yaml

Il file **values.yaml** del PACKS package è il cuore del progetto dello studente. Fortemente ispirato a Docker Compose, permette di centralizzare in un unico file di configurazione dichiarativa tutte le applicazioni containerizzate, per un rilascio su un ambiente distribuito Kubernetes invece che in locale.
I "components" indicano appunto le componenti, quindi i container, che formano l'applicativo d'insieme. Si tratta di una lista dove ogni elemento contiene una serie di attributi "chiave:valore" per la configurazione interna.

6.1.2.1 Container basilare

Ogni componente deve avere un campo "name", un campo "image" ed un parametro "active" per essere definito tale. L'immagine deve appartenere ad un registry conosciuto come DockerHub dato che, allo stato dell'arte, non è possibile eseguire una "build" di un Dockerfile. Però, come si vedrà, sarà possibile creare un container tale e quale ad esso grazie ai vari tipi di configurazioni permesse dal PACKS Package.
Il campo "active" contiene un valore booleano che indica lo stato desiderato della componente che, se impostata su "false", viene ignorata dal PACKS template e sarà avviata insieme al resto del progetto utente. In pratica tutte

le componenti fondamentali debbono essere impostate su "active: true" per poter essere monitorate e seguite attraverso la UI.

```
components:
- name: python
  image: python:latest
  active: true
```

Per inserire altre componenti basta aggiungere un altra voce alla lista con il trittico di informazioni minime sopra descritte.

```
components:
- name: python
  image: python:latest
  active: true
  ports:
  - port: 8080
- name: nginx
  image: nginx:latest
  active: true
  ports:
  - port: 9090
```

Con un values.yaml del genere è gia possibile effettuarne l'upload su PACKS UI e la visualizzazione dei primi logs basilari. Se l'interfaccia web di upload restituirà errore bisogna ricontrollare la sintassi e verificarne la correttezza.

6.1.2.2 Ports

Per indicare le porte esposte dai container, cosi come Docker Compose e Kubernetes, viene in aiuto il campo "ports". Questo contiene una lista dove ogni elemento possiede l'attributo fondamentale "port: numero_intero" che indica la porta esposta interna al container, così come fa il layer EXPOSE in un Dockerfile. Il campo ausiliario "protocol" rende possibile specificare il protocollo di comunicazione accettato (TCP o UDP). Una funzione importante è affidata al parametro con chiave "hostPort" poichè permette di indicare una porta esposta sul nodo Kubernetes interno per l'accesso diretto, in breve crea un NodePort. Per limitazioni Kubernetes, il numero di porta hostPort deve essere un intero compreso fra 30000 e 32767.
Questa sezione indicava alla UI che il container avesse una porta accedibile dall'esterno ed infatti tale implementazione sarà pian piano deprecata durante lo sviluppo del reverse proxy server interno, il quale può gestire automaticamente il routing senza esposizione diretta sull'ambiente sottostante. Anche perchè, fra più progetti, è estremamente probabile che prima o poi venga occupata una porta con quel particolare numero sul nodo e quindi

l'insorgere di problemi e conflitti diverrebbe esponenziale e darebbe un pessimo livello di isolamento interno. In futuro la voce "hostPort" sarà sostituita con un semplice valore booleano per indicarne il possibile accesso dalla UI.

```
ports:
- port: 8080
  protocol: TCP
  hostPort: 30000
- port: 9090
  hostPort: 30001
```

6.1.2.3 Environment

Questo campo si occupa semplicemente di settare le variabili d'ambiente attraverso una lista di elementi formati da due parametri minimi: "name" e "value". Si puo intuire facilemente il loro ruolo, infatti "name" è il nome della variabile e "value" ne indica il contenuto, che esso sia un numero o una stringa.

```
environment:
- name: ENV_VAR_NAME
  value: 12345
- name: JAVA_HOME
  value: /path/to/some/folder
```

6.1.2.4 Commands

Si tratta di un parametro che, sempre attraverso una lista, fornisce la possibilità di elencare una serie di comandi bash da eseguire all'interno del container. Essi verranno avviati uno per uno nell'ordine di definizione. La sintassi della lista per il values.yaml è la seguente:

```
commands:
- command: cd $HOME/folder
- command: python program.py
```

Bisogna stare molto attenti nell'utilizzo dei commmands del file values, perchè vi è possibilità che questi vadano a sostituire l'Entrypoint dell'immagine container predefinita.

6.1.2.5 Volumes

Questa sezione è completamente ispirata a Docker Compose, soprattutto nel modo in cui si vogliono montare le cartelle ed i file all'interno del container. Cambia invece la sintassi, diventando più prolissa. Ogni elemento della lista "volumes" deve avere il campo "name" ed il campo "mountPath". Il primo è

autoesplicativo, il secondo definisce il percorso di montaggio interno. Questo valore cambia in base al tipo di volume poichè si può differenziare fra intera directory e singolo file. Infatti se si tratta di un file, il percorso di origine verrà indicato nel parametro "file", se invece si tratta di una directory sarà necessario il parametro "directory" per indicarne la posizione originale. Di seguito un esempio di campo "volumes" e le sue possibili configurazioni:

```
volumes:
- name: volume-name
  directory: /path/to/the/folder/to/mount
  mountPath: /path/to/the/mount/destination/folder
- name: volume2-name
  file: /path/tothefileto/mount.conf
  mountPath: /path/to/the/mount/destination/file.conf
```

Una particolare attenzione va impiegata nella scrittura del percorso di origine dei file o delle cartelle. Esso segna il "path" a partire dall'interno del file .zip in allegato al package del progetto. Quindi se PACKS deve avviare un container che vuole montare il volume "volume2-name", dell'esempio sopra, con percorso di origine "/path/tothefileto/mount.conf", si aspetterà di trovare quel file all'interno dello zip con il seguente contenuto interno, nell'esatto ordine:

```
|- components.zip
|    - /path/
|    |    - /tothefileto/
|    |    |    - mount.conf
|
|    - /otherDirectoriesAndFiles/
     ...
...
```

6.1.2.6 Jobs

Un potenziamento del campo "commands" è fornito in "Jobs". Ispirato all'oggetto Job di Kubernetes, permette di eseguire la lista dei comandi forniti al suo interno in un container con un immagine uguale a quella principale, o del tutto separata. La sintassi richiama i precedenti campi "image" e "commands" e li unisce sotto un unica voce "jobs". Un esempio è il seguente:

```
jobs:
  image: image-that-will-run-the-command
  commands:
  - command: bash command 1
  - command: bash command 2
```

Campo molto utile quando si vuole configurare un altro container dall'esterno attraverso richieste di rete.

6.1.3 Zip file

Nel paragrafo precedente è stata descritta la componente fondamentale per la creazione del PACKS package. Allo stato dell'arte, per montare un volume con una directory o un file specifico nei vari container, non basta soltanto specificare i percorsi ed il tipo sul values.yaml ma bisogna anche fornire i file citati. Quindi il campo "volumes" senza gli elementi da montare è piuttosto inutile e v'è bisogno di un modo per fornirli al sistema. Un file con estensione zip conterrà, in un formato compresso, tutti questi elementi.
Come accennato prima, non basta soltanto l'inserimento di file o cartelle di script o configurazioni ma bisogna farlo seguendo il percorso di origine indicato in "volumes", come negli esempi 6.1.2.5 e 6.1.2.5.
Questo è un limite dovuto a Kubernetes poichè molti file sarebbero passabili e caricabili via ConfigMap, però la dimensione massima imposta da etcd è di 1 Megabyte e la suddivisione dei file in più elementi ne renderebbe complicata la gestione. Una soluzione migliore si avrebbe con l'uso di un Object Storage, sia Cloud, come Amazon S3, sia interno, come quello che forniscono diversi programmi opensource.

6.2 Helm PACKS Template

Una volta creati e caricati il file "values.yaml" ed il file zip in PACKS UI, come descritto in precedenza, vengono inoltrati a Helm-Manager che si occupa della loro installazione.
All'avvio del progetto utente, attraverso un client Helm e l'oggetto "action.Install", forniti dalle librerie ufficiali in Go, avviene la creazione di un Chart con un template specifico che prende in input il file yaml del PACKS package, il nuovo nome e il namespace (entrambi token JWT). Il template predefinito, usato per ogni progetto o caricamento dallo studente, è colui che permette l'adattamento e la standardizzazione di un file yaml, simile al Compose, in oggetti Kubernetes. Il motore di templating offerto da Helm è centrale per questa "traduzione", specialmente date le logiche di inserimento contenenti funzioni, cicli e blocchi condizionali.
Un sistema simile che svolge una conversione da Docker Compose a Chart Helm esiste già da qualche tempo e si chiama "Kompose". Ma l'utilizzo di questo strumento presume una conoscenza da parte dello studente non solo dei manifest Kubernetes ma anche di Helm. Inoltre i chart creati sono molto generalizzati e spesso ricorrono in errori di imprecisione nella conversione. Oltre alle motivazioni appena elencate, Kompose è stato scartato anche perchè è un tool a riga di comando non semplice da integrare in Helm Manager e

di conseguenza in PACKS. Quindi è stato preferito l'utilizzo di un unico file "template.yaml", il quale si adatta meglio e con meno errori sia al deployment di insiemi di applicazione containerizzate, sia all'integrazione con le librerie Go di Helm e fornisce un buon livello di astrazione delle componenti K8s.

6.2.1 Prodotti conversione

Il template crea principalmente, a partire dal values.yaml, due risorse Kubernetes: un Deployment ed un Service ad esso associato.
Attraverso l'esclusivo utilizzo di almeno 2 elementi si mantiene una semplicità implementativa efficace per la gestione dei container. Trattandosi di sistemi containerizzati non destinati ad un ambiente di produzione, non vi è la necessità di sfruttare le proprietà di High Availability offerte da Kubernetes. Il Deployment gestirà un ReplicaSet ad un elemento che, a sua volta, conterrà un Pod. Il container definitovi all'interno avrà tutte le specifiche desiderate nel values.yaml settate secondo la corretta sintassi del manifest. I volumi saranno montati correttamente con un percorso che parte dalla root del nodo sottostante fino ad arrivare alla cartella d'installazione, nel quale vi sono estratti i contenuti del file zip passato in upload. Quindi è fondamentale, per questo motivo, la condivisione della cartella "/shared/uploads/" fra i nodi del cluster poichè un container che si basa su dei file montati, ovunque esso sia eseguito, deve accedervi indiscriminatamente per il corretto funzionamento.
Il Service viene creato automaticamente alla presenza di porte esposte nel file values e ne viene definito il tipo in base alle caratteristiche. In genere è un ClusterIP ma se una porta è hostPort allora il tipo cambia in NodePort. Come già detto in precedenza questo sistema verrà deprecato e tutti i Service saranno esclusivamente del tipo di default, cosa che renderà più semplice anche il template.
Oltre alle due risorse fondamentali ve n'è una terza creata solo in presenza di Jobs nel file yaml utente. Si tratta appunto dell'oggetto Jobs di Kubernetes, ossia un Pod che ha un container, il quale esegue un comando specifico, che sia di configurazione o di debug per le altri componenti, destinato a terminare con successo. Infatti l'istanza Jobs che controlla il Pod, se questo termina con errore e non effettua correttamente il comando, viene riavviata per un certo numero di volte fin quando non si raggiungono il numero di tentavi massimi permessi oppure il successo del compito.

6.2.2 Implementazione

Il "template.yaml" è un manifest Kubernetes a tutti gli effetti che contiene blocchi di codice del template engine di Helm. Ogni blocco, che sia condiziale oppure di un ciclio iterativo, si apre con le doppie parentesi graffe, contenenti la funzione o il valore e si chiude attraverso la parola chiave " end ". Prima di tutto, utilizzando l'oggetto ".Values" messo a disposizione da Helm, si vanno a leggere tutti i valori del "values.yaml" caricato dallo studente. Questi, una volta letti, vengono assegnati ad una variabile globale "$globalValue", valida per tutto il documento.

```
1     {{ $globalValue := .Values }}
```

Successivamente vi sono due grossi blocchi di codice. Il primo utilizza la funzione "range" per iterare attraverso i vari "components" del file yaml, in breve sarebbe uguale ad un "forEach" come funzionalità. Il secondo è un condizionale "if" che controlla se la componente ha il parametro "active" settato su "true". In poche parole, da questo momento avviene l'iterazione container per container (o component per component), definendone gli elementi Kubernetes se settato come attivo.

6.2.2.1 Creazione Deployment

Al nuovo deployment vengono assegnati il nome della componente affiancato dalla stringa "-deployment" per il campo "name" nei metadati ed il campo "app" di "labels". Il ReplicaSet definito dalla prima chiave "spec" definisce il numero di repliche, se non specificato nel values.yaml, ad una attraverso una seplice pipeline:

```
1 ...
2 spec:
3   replicas: {{ .replicas | default 1 }}
4 ...
```

La funzione "default", come già trattato precedentemente, scrivi il valore in input se non esiste quello prima.
Il selettore Pod del ReplicaSet deve avere le stesse "labels" del Pod in "matchLabels" per potervisi associare correttamente, quindi qui si riutilizza la stringa con il valore "name" con "-deployment" postfisso nel campo "app". Segue lo stesso sistema di nomenclatura ed edichette il Pod sottostante definito nel campo "template". La complessità configurativa è presente tutta nel parametro "spec" del Pod, dove si definiscono il container ed eventuali volumi.
Un blocco "if" cotrolla la presenza di porte esposte, ossia valori della lista "ports" per il settaggio di un hostname. A questo punto si trascrivono nei

campi appositi l'immagine ed il nome del container. Se presenti porte da esporre, un ciclo "range" itera su di esse settando i vari "containerPort" e relativo nome. QUest'ultimo in particolare viene creato, attraverso delle funzioni in pipeline, a partire dal numero di porta, a cui si aggiunge il protocollo.

```
...
    containers:
      - name: {{ .name }}-container
        image: {{ .image }}
        {{ if .ports }}
        ports:
        {{ range .ports }}
        - name: {{ .port }}-{{ .protocol | lower | default "
            port" }}
          containerPort: {{ .port }}
        {{ end }}
        {{ end }}
```

Se presenti variabili d'ambiente vengono inserite secondo la corretta sintassi dentro un blocco "if", che ne controlla la presenza, ed un "range" che le itera. I valori vengono passati alla funzione "quote" che li inserisce in delle virgolette indicando al manifest di trattarle come stringhe.

```
...
{{- if .environment }}
    env:
    {{ range .environment }}
    - name: {{ .name }}
      value: {{ .value | quote }}
    {{- end }}
{{- end }}
```

I "commands", se presenti, vengono inseriti come argomenti del comando principale "/bin/bash -c" ed elencati sempre attraverso un ciclo "range". La scelta di questo formato dovrebbe garantire la possibilità di una corretta esecuzione di comandi in serie all'interno del container.

```
...
    {{- if .commands }}
        command: ["/bin/bash", "-c"]
        args:
        - |
            {{ range .commands }}
            {{ .command }}
            {{ end }}
    {{ end }}
```

Un interesse particolare è richiesto per la definizione dei "volumeMounts". Racchiusi in un condizionale "if" che ne controlla la presenza, il ciclo itera su

di essi assegnando "name", "mountPath" e "subPath". Quest'ultimo solo se il tipo di volume è un file poichè il campo Kubernetes permette il montaggio di un file unico in questo modo.

```
...
    {{- if .volumes }}
        volumeMounts:
        {{- range .volumes }}
        - name: {{ .name }}
          mountPath: {{ .mountPath }}
          {{- if .file }}
          {{- $filename := base .file }}
          subPath: {{ $filename }}
          {{ end }}
        {{- end }}
    {{- end }}
```

Terminata la definizione del container, se vi sono stati definiti dei volumeMounts, bisogna inserire necessariamente anche i volumi nel secondo campo all'interno di "spec". I volumi sono definiti nel campo "volumes" ed attraverso una iterazione vengono elencati assegnandovi nome e "hostPath". Alla variabile ausiliara "sourcePath" viene affidato il valore del campo "file" o del campo "directory" se presenti in "values.yaml". Alla chiave interna "path", attraverso una serie di funzioni, viene assegnato il valore di "sourcePath" pulito e formattato correttamente.

```
...
    volumes:
      {{- range .volumes }}
      - name: {{ .name }}
      {{- $sourcePath := "" }}
         {{- $sourcePath = .directory }}
         {{- if .file }}
           {{- $sourcePath = .file }}
         {{- end }}
        hostPath:
          path: /{{ $globalValue.rootDirectory }}{{ if .file
              }}{{ dir $sourcePath }}{{ else }}{{ $sourcePath
              }}{{ end }}
      {{- end }}
      {{- end }}
```

6.2.2.2 Creazione Service

Il "template.yaml" dopo aver definito il Deployment per il progetto in base ai valori passati nel file "values.yaml", se vi sono presenti delle porte esposte, crea un Service per fornire loro una interfaccia di rete statica.

All'interno di un blocco "if" vi è il manifest del Service. Nei metadati, il parametro "name" assume il valore dell'omonimo nel component. Nel campo "spec", in genere, vengono definiti il tipo di Service, il selettore e le porte da inoltrare. Come prima cosa un ciclo "range" itera fra le porte fornite nel values.yaml e se ne trova una con la voce "hostPort", ne imposta il tipo in NodePort. Il selettore invece viene impostato con etichette uguali al Deployment appena creato in modo tale che vi si possa associare. La sezione "ports", cosi come l'omonima nel file yaml utente, contiene una lista. Ogni elemento di questa lista deve avere: un nome, il protocollo, la porta da esporre, la porta a cui inoltrare i pacchetti e l'eventuale porta nodePort da montare sulla macchina host. Il templating di Helm compila questi valori iterando sulla lista di values, creando il nome in base alla combinazione della porta e del protocollo ed inserendo il protocollo fornito, oppure quello di default (TCP). Successivamente assegna gli stessi valori sia alla voce "port" che alla "targetPort" e, se vi è un campo "hostPort", ne assegna il valore a "nodePort".

```
1 ...
2 spec:
3   {{ range .ports }}
4   {{ if .hostPort }}
5   type: NodePort
6   {{ end }}
7   {{ end }}
8   selector:
9     app: {{ .name }}-deployment
10  ports:
11  {{- range .ports }}
12  - name: {{ .port }}-{{ .protocol |  lower | default "port"
      }}
13    protocol: {{ .protocol | default "TCP" }}
14    port: {{ .port }}
15    targetPort: {{ .port }}
16    {{ if .hostPort }}
17    nodePort: {{ .hostPort }}
18    {{ end }}
19  {{ end }}
```

6.2.2.3 Costruzione Job

Se specificato nel file YAML fornito dall'utente, l'ultimo elemento Kubernetes associato al container del progetto sarà il Job. La nomenclatura e i metadati seguono le stesse regole già descritte per gli altri oggetti Kubernetes, con l'unica differenza che al campo "name" viene aggiunto il suffisso "-job".

Il Pod specificato all'interno della chiave "spec.template" del Job presenta una struttura simile a quella di un Pod standard, con metadati che rispecchiano quelli del Job stesso. Tuttavia, il parametro "restartPolicy" è impostato su "OnFailure", il che significa che il Pod verrà riavviato automaticamente solo in caso di fallimento dell'esecuzione.

La definizione dei container all'interno del Job ne prevede uno soltanto, la cui immagine viene indicata nel file values.yaml. Per quanto riguarda il campo "command", questo segue la stessa impostazione utilizzata nei Pod del Deployment. La principale differenza risiede nel campo "args", dove viene iterata e stampata la lista di comandi specificati sotto un unica grande stringa."

```
1 ...
2 template:
3     metadata:
4       labels:
5         app: {{ .name }}-job
6     spec:
7       restartPolicy: OnFailure
8       containers:
9       - name: {{ .name }}-job
10        image: {{ .jobs.image}}
11        command: ["/bin/bash", "-c"]
12        args:
13        - |
14        {{- range .jobs.commands }}
15            {{ .command }}
16        {{- end }}
```

Il file completo "template.yaml" si trova all'interno della cartella dei file sorgente di Helm-Manager.

Codice 6.2: esempio generale della sintassi values.yaml

```
components:
- name: ubuntu
  image: ubuntu:latest
  active: true
  ports:
  - port: 8080
    hostPort: 30001
  - port: 9090
  environment:
  - name: SOME_NUMBER_VAR
     value: 89
  - name: ANOTHER_ENV_VAR
     value: hola
  commands:
  - command: apt get update
  - command: apt install nginx
  volumes:
  - name: volume-name
    directory: /path/to/the/folder/to/mount
    mountPath: /path/to/the/mount/destination/folder
  - name: volume2-name
    file: /path/to/the/file/to/mount.conf
    mountPath: /path/to/the/mount/destination/file.conf
  jobs:
    image: job-image:latest
    commands:
    - command: command that interact with another component
    - command: another bash command

- name: another-component
  image: another-component:latest
  .......
  ........
```

Conclusione

6.3 Riepilogo

Il progetto nasce dall'esigenza di semplificare la gestione e il deployment di container in ambienti Kubernetes, in particolare, per studenti universitari che spesso non dispongono di risorse hardware adeguate o competenze avanzate per gestire infrastrutture distribuite complesse. L'obiettivo principale è offrire una piattaforma che permetta di concentrarsi sullo sviluppo del codice e delle applicazioni containerizzate, senza dover affrontare direttamente le complessità legate alla configurazione, scalabilità e networking.
Oltre alla carenza di risorse, i requisiti che PACKS vuole possedere sono vari. Risulta essenziale garantire una soluzione che semplifica il processo di testing e di verifica dei risultati da parte del docente responsabile dell'insegnamento, mentre fornisce, similmente ad un ambiente Cloud CaaS personale, una piattaforma semplificata che permette gestire in maniera centralizzata la creazione, lo sviluppo ed il perfezionamento di applicativi clusterizzati.
Tutto questo deve essere accompagnato da un buon livello di efficienza, di autenticazione e di sicurezza.

6.4 Obiettivi raggiunti

I requisiti funzionali iniziali sono stati quasi tutti soddisfatti. La creazione di un'interfaccia utente (UI) intuitiva ha semplificato notevolmente lo studio e la gestione dei container, concentrando tutte le operazioni in pochi semplici passaggi. Questo ha reso l'interazione con il sistema molto più accessibile. Inoltre, il sistema di autenticazione basato su token JWT ha garantito che ogni richiesta e comunicazione tra le varie componenti fosse autenticata, migliorando significativamente la sicurezza complessiva della piattaforma.
Un ulteriore passo avanti è stato fatto con l'introduzione di un formato di packaging "standard" per il sistema, dove l'uso di una sintassi semplice ha contribuito a formare un livello di astrazione che nasconde la complessità di

Kubernetes e Helm, pur mantenendo intatto l'approccio dichiarativo, utile per l'apprendimento di questi sistemi di orchestrazione dei container.
La possibilità di visualizzare lo standard output e di gestire il forwarding delle porte esposte consente agli utenti di evitare, in gran parte, l'uso estensivo delle istruzioni da riga di comando. Questo facilita notevolmente l'esperienza d'uso, specialmente per chi non ha grande familiarità con i comandi CLI.
In termini di sicurezza, il sistema di consegna dei task tramite token monouso protegge il docente amministratore da possibili abusi, garantendo un uso corretto delle funzionalità offerte.
Un aspetto particolarmente impegnativo del progetto è stato il soddisfacimento dei requisiti non funzionali. Il sistema di gestione dei token, come già accennato, è stato complesso da progettare e implementare, e la sua piena potenzialità non è ancora del tutto sfruttata. D'altro canto, l'integrazione con il sistema di login basato sulle credenziali universitarie memorizzate nel server LDAP rappresenta un primo passo solido per la gestione dell'autenticazione iniziale. Questo approccio semplifica la gestione degli account per gli studenti, eliminando la necessità di registrazioni multiple e riducendo potenziali complicazioni legate alla creazione e gestione degli account.
PACKS è in grado di gestire un buon numero di utenti e deployment simultanei. La conversione dei container in Deployment e Service consente a Kubernetes di distribuire in modo efficiente il carico tra i vari nodi del cluster. Inoltre, poiché le componenti di PACKS operano all'interno della stessa infrastruttura, è possibile replicare i Pod del sistema in base alle necessità, scalando verticalmente per rispondere a un incremento della domanda. Se le risorse hardware centrali dei server non dovessero bastare, il "join" di un altro nodo risolverebbe all'istante il problema.
I limiti sulle dimensioni e sulla correttezza dei file caricati dallo studente contribuiscono ad innalzare il livello di affidabilità del sistema, evitando inutili abusi di risorse.
Tutto sommato, il progetto riesce a fornire un primo approccio funzionante alla soluzione del problema che attanaglia gli studenti con poche risorse hardware. Ragionando con astrazioni si è riusciti a dare un servizio semplice ma che lascia la giusta complessità didatticalmente utile allo studente per imparare ed al professore per valutarne il lavoro.

6.5 Limiti e sviluppi futuri

Come accade spesso nei sistemi giovani, anche PACKS potrebbe presentare alcuni bug o malfunzionamenti dovuti alla sua recente introduzione. Tuttavia, ciò che conta è il percorso che verrà intrapreso attraverso il mantenimen-

to, lo sviluppo continuo e il miglioramento costante, elementi fondamentali per l'evoluzione di un software di qualità.

Uno dei principali limiti iniziali di PACKS riguarda la quantità di nodi worker Kubernetes. Quando uno studente trasferisce l'esecuzione del proprio progetto da un PC a PACKS, il sistema deve disporre di risorse superiori a quelle del singolo computer, soprattutto considerando la potenziale ampia utenza universitaria. Una soluzione razionale potrebbe includere non solo l'espansione delle capacità di calcolo e lo scaling orizzontale, ma anche l'integrazione di un sistema di monitoraggio ("metrics") che misuri costantemente l'utilizzo delle risorse da parte dei container. Kubernetes fornisce già delle librerie Go per implementare un sistema di metriche basilare, per cui l'aggiunta di questa funzionalità potrebbe essere un futuro sviluppo interessante.

Attualmente, tutte le comunicazioni tra le componenti avvengono tramite HTTP non cifrato. Sebbene sia presente un token di autenticazione cifrato, esiste ancora il rischio di attacchi "replay". L'adozione di HTTPS per la comunicazione sia interna al cluster che esterna (ad esempio, verso la UI di PACKS) sarà cruciale, soprattutto in un ambiente di produzione.

Redis, che al momento gestisce tutte le informazioni sensibili, deve essere ulteriormente protetto. L'accesso libero ai dati deve essere abolito, e sarà obbligatorio configurare un account protetto da password per garantire la sicurezza delle informazioni.

Una volta raggiunta una maggiore stabilità e sicurezza, si potrebbe implementare una funzionalità che consenta la gestione individuale dei container. Ciò permetterebbe agli utenti di avviare o fermare i Pod singolarmente e modificare i singoli componenti del file values.yaml, migliorando la flessibilità e il controllo.

Un'altra possibile ottimizzazione riguarda la semplificazione del template Kubernetes e l'eliminazione dell'uso di Service NodePort per i progetti utente. Questo ridurrebbe i conflitti sulle porte dei nodi e migliorerebbe l'isolamento dei container.

La struttura interna dei Pod di PACKS potrebbe essere ulteriormente migliorata. Ad esempio, l'attuale utilizzo di un DaemonSet per il reverse proxy Nginx potrebbe essere sostituito da un Deployment, che si adatta meglio alle esigenze del sistema. Inoltre, poiché le componenti di PACKS sono progettate per scalare tramite repliche attive, l'introduzione di un Horizontal Pod Autoscaler automatizzerebbe il processo di scalabilità basandosi sulle richieste o su metriche personalizzate.

Infine, il livello di autenticazione potrebbe essere potenziato implementando la verifica delle firme dei token da parte di tutte le componenti del sistema, non solo nella UI. Al momento, le altre componenti si limitano a

confermare la presenza del token in Redis, ma una verifica più completa garantirebbe maggiore sicurezza.

6.6 Considerazioni finali

Le potenzialità di PACKS sono immense, grazie anche alla flessibilità e all'ampia gamma di librerie e package offerti da NodeJS e Go, due tecnologie che sono al cuore del progetto. Le librerie ufficiali usate hanno enormi potenzialità, essendo molto vaste e largamente supportate. Questo rappresenta un punto di forza per PACKS, che può contare su delle tecnologie all'avanguardia per affrontare le prossime sfide.
Il raggiungimento di un livello elevato di maturità e la realizzazione di standard ottimali per PACKS non sono che una questione di tempo e di un costante lavoro di ottimizzazione. Un elemento cruciale in questo processo sarà l'implementazione di una solida pipeline CI/CD (Continuous Integration/Continuous Delivery). Grazie a tale infrastruttura sarà possibile accelerare il ciclo di vita dello sviluppo, migliorare la qualità del codice e ridurre i tempi di rilascio. La CI/CD non solo automatizzerà il processo di testing e integrazione del software, ma permetterà anche una distribuzione continua e priva di interruzioni, mantenendo l'efficienza operativa del sistema. Essendo un Chart Helm il risultato d'insieme, la distribuzione di nuove versioni potrà essere più immediata e semplice.
In conclusione, se da un lato PACKS ha già raggiunto una base solida e funzionante, dall'altro ci sono molte possibilità evolutive. Grazie a delle buone fondamenta, il progetto ha davanti a sé un bel futuro. Con il giusto approccio allo sviluppo continuo, il coinvolgimento della comunità e l'attenzione ai dettagli, PACKS potrà diventare una soluzione di riferimento per la gestione e lo studio di sistemi containerizzati in ambito universitario, contribuendo all'apprendimento di numerosi studenti.

Bibliografia

[1] Google Cloud. Definizione di container, 2024. Accesso: 24 agosto 2024.

[2] Wallarm. High availability cluster - workflow, 2024. Accesso: 21 agosto 2024.

[3] RedHat. Saas architecture, 2024. Accesso: 21 agosto 2024.

[4] Kubernetes. Components of kubernetes, 2024. Accesso: 21 agosto 2024.

[5] Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson. *Kubernetes Guida pratica*. O'Reilly, LegoDigit Srl, TN, 2023. Capitolo: Pod in Kubernetes.