UNIVERSITÁ POLITECNICA DELLE MARCHE

FACOLTÁ DI INGEGNERIA

Corso di Laurea Triennale in
Ingegneria Informatica e dell'Automazione

Riprogettazione di un ERP mediante container Docker ed orchestrazione dei servizi mediante Kubernetes

Redesign of an ERP system through Docker containers and services orchestration using Kubernetes

Relatore:
Dott. Mancini Adriano

Laureando:
Licci Giacomo

Anno Accademico 2019-2020

Indice

Capitolo 1

Introduzione

1.1 Obiettivo

Il progetto di tesi si concentra sul riprogettare un'applicazione web con architettura ERP *(Enterprise Resource Planning)* utilizzando macchine virtuali leggere denominate *Container* per poi gestire le rispettive immagini con un sistema di orchestrazione dei servizi utilizzando Kubernetes. Il tutto viene realizzato attraverso un servizio di *Cloud Computing*, in modo tale che l'applicazione funzioni correttamente e in sicurezza.
All'interno della maggior parte delle aziende, capita molto spesso di trovare un ERP come sistema di gestione in grado di integrare tutti i processi di *business* più importanti quali contabilità, vendite, acquisti ecc. Ed è proprio grazie a questa spina dorsale informatica che l'azienda è in grado di ottimizzare tutti i processi aziendali eliminando sprechi ed errori, e quindi, ottenere spese e costi ridotti per l'impresa [10].
Il progetto di tesi è stato svolto a seguito del tirocinio presso l'azienda Meta Informatica S.R.L. nella provincia di Pesaro e Urbino [17]. La suddetta azienda si occupa di promuovere e distribuire ad altre aziende, un *software* ERP con denominato Bike, fornendo assistenza nel suo utilizzo. Il progetto consiste nel ricreare Bike utilizzando 2 container tramite Docker: nel primo sarà ricreata la base di dati e il secondo ospiterà l'applicazione web. Una volta aver testato il loro funzionamento in locale verranno salvati come immagini Docker che saranno poi caricate su Google Cloud Platform, tramite il servizio di Google Kubernetes Engine verrà creato un Cluster dove saranno eseguiti i container precedentemente creati tramite strumenti che permettono il funzionamento corretto e costante dell' applicazione correggendo errori che potrebbero insorgere nell'utilizzo.

1.2 Struttura della Tesi

La parte principale del progetto di tesi è il deploy del *software* ERP Bike (Approfondimento Capitolo 3) utilizzando container creati tramite Docker residenti all'interno di Cluster. Partiremo, quindi, con l'introdurre il concetto di Container, Cluster. Successivamente verranno introdotti Docker e Kubernetes insieme ai rispettivi servizi; a seguito di ciò verrà definito il sistema ERP e la struttura di Bike. A questo punto verrà esposto tutto lo sviluppo dell'applicazione partendo dall'analisi dei requisiti e arrivando al *testing* indicando anche le eventuali problematiche incontrate indicando, infine, possibili sviluppi futuri. La tesi è organizzata nel seguente modo:

- Nel capitolo 2 viene introdotto il concetto di Container, Cluster e delle loro rispettive applicazioni Docker e Kubernetes tramite il servizio di Google Coud Platform: Google Kubernetes Engine;
- Nel capitolo 3 sono definiti il funzionamento e la struttura di un sistema ERP indicando i vantaggi del suo utilizzo, procedendo con la descrizione del *software* Bike e la sua architettura;
- Il capitolo 4 riporta lo sviluppo dell'applicazione, nel dettaglio verrà svolta un analisi dei requisiti per creare i container per poi procedere con l'implementazione e il test su Cluster indicando, infine, le varie problematiche affrontate;
- Nel capitolo 5 sono esposte le conclusioni e possibili sviluppi futuri del progetto di tesi.

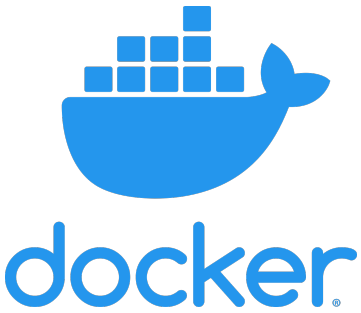
Capitolo 2

Docker e Kubernetes

2.1 Docker

Docker è un progetto *open-source* che consente l' installazione e l'esecuzione, di una o più applicazioni all'interno di contenitori (*container*) *software*. Rilasciato per la prima volta il 13 marzo 2013 Docker nacque per lavorare su Kernel Linux ma nell'ottobre 2014 Microsoft annunciò il supporto nativo su Windows Server 2016. La prima versione stabile venne rilasciata a fine Luglio 2016, dando la possibilità di creare container su Kernel Windows. [7] I container Linux creati in ambiente Windows vengono virtualizzati attraverso Hyper-V o Oracle VirtualBox non potendo eseguirli direttamente su Kernel Windows.
Il funzionamento di Docker consiste nell'utilizzare il *Kernel* di Linux (o Windows) e le sue risorse, per isolare processi e applicazioni ed eseguirli in modo indipendente e separati tra di loro, evitando l'installazione e la manutenzione di una macchina virtuale intera, ottimizzando l'uso dell'infrastruttura utilizzata e conservando il livello di sicurezza garantito dalla presenza di sistemi separati [9].

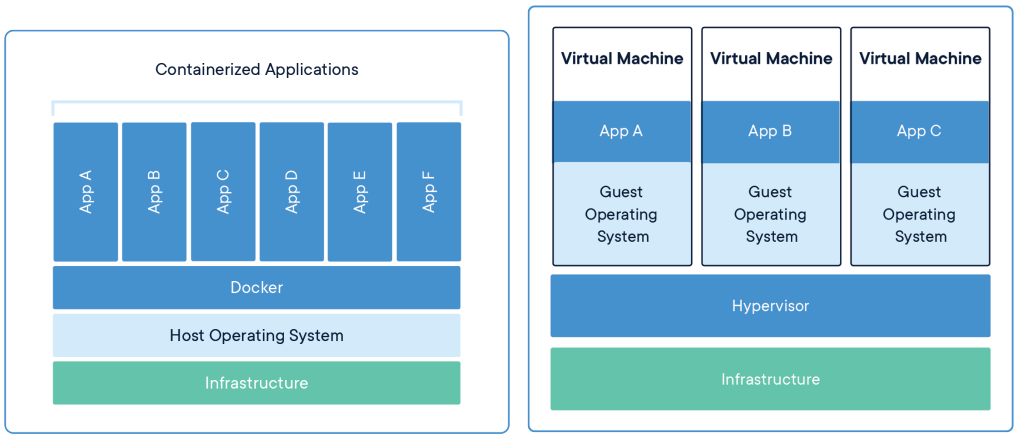
Figura 2.1: Logo di Docker



2.1.1 Cos'è un Container

Per Container si intende una unità *software* contenente librerie, rami di *filesystem*, *file* di configurazione e altri eseguibili per il corretto funzionamento di un' applicazione ed é in grado di velocizzare l'esecuzione dell' applicazione, rendendola affidabile a seconda dell'ambiente di calcolo su cui risiede [22].
All'apparenza i Container potrebbero essere accomunate a macchine virtuali, ma ci sono sostanziali differenze: se le macchine virtuali sono un emulazione a livello *Hardware* di una o più macchine fisiche, ognuna con un sistema operativo completo, i container sono un "emulazione" a livello *Applicazione* contenenti un insieme di dati per l'esecuzione di un'applicazione (Figura 2.2). Possono essere si eseguiti nella stessa macchina fisica ma condividono lo stesso Kernel del sistema operativo, come se fossero processi isolati, comportando una minor occupazione di spazio (qualche MB) e dando la possibilità di eseguire più applicazioni senza affaticare troppo il sistema operativo, a differenza delle MV che, oltre ad essere molto lente in fase di avvio, richiedono molte risorse (diversi GB) per virtualizzare l'hardware e creare un sistema operativo interno [6].
Le applicazioni "containerizzate", con base Linux o Windows a seconda dei casi, vengono eseguite nella stessa maniera indipendentemente dall'infrastruttura grazie all'isolamento che i container creati attuano tra applicazione e macchina fisica. Anche cambiando infrastruttura l'esecuzione dei processi rimane identica evitando così problemi di incompatibilità come spesso accade con applicazioni normali. Docker offre degli strumenti utili sviluppati da container Linux con cui si possono eseguire *deploy* rapidi, e controllare la distribuzione di nuove versioni, rivelandosi uno strumento semplice per creare e gestire container [9].

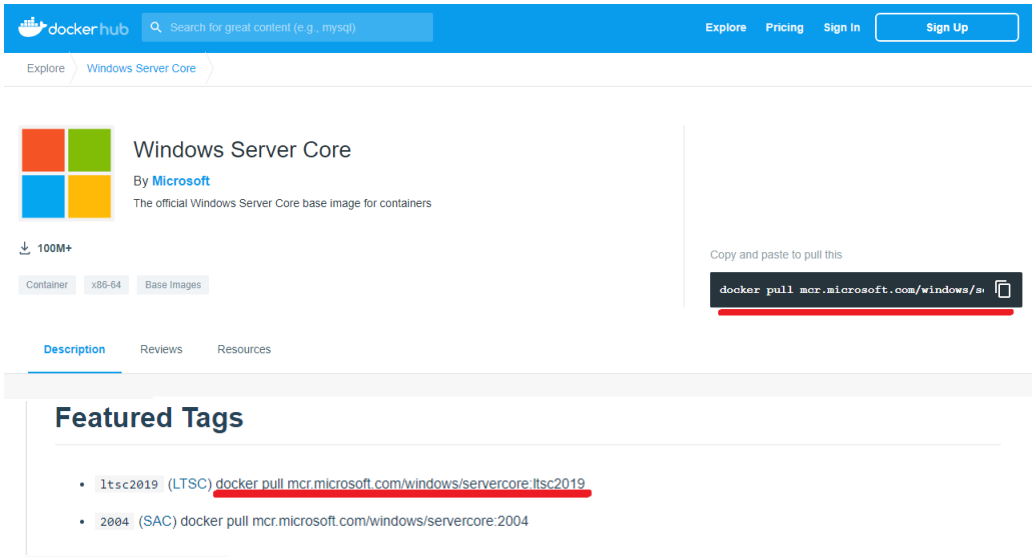
Figura 2.2: Differenza tra Container e MV



2.1.2 Immagini Docker e Dockerfile

Le immagini Docker sono dei pacchetti *software* indipendenti contenenti codice, strumenti e librerie di sistema con rispettive impostazioni, i quali permettono l'esecuzione dell'applicazione su *Docker Engine* e diventando container in stato di esecuzione. Creare un container con gli strumenti di Docker, partendo da un'immagine, semplifica la condivisione di un'applicazione o di un insieme di servizi nei vari ambienti, mantenendo tutte le loro dipendenze. Docker automatizza anche la distribuzione dell'applicazione o processi all'interno dell'ambiente "containerizzato".
Le immagini Docker base sono disponibili all'interno del *Docker Hub* (Fig. 2.3) scaricabili tramite il comando `docker pull` specificando il nome dell'immagine oppure vengono create utilizzando un documento di testo denominato *Dockerfile* contenente tutte le istruzioni a linea di comando per poter creare un'immagine ad hoc per il suo utilizzo. Tramite il comando `docker build` tutte le istruzioni contenute nel Dockerfile vengono eseguite in successione automaticamente dando anche la possibilità di dare un nome all'immagine tramite l'opzione `-t` [8].

Figura 2.3: Docker Hub



Tra le istruzioni principali utilizzate all'interno del Dockerfile troviamo:

- `FROM <nome immagine>`:È la prima istruzione che viene eseguita, inizializza lo stadio di *build* impostando l'immagine selezionata come immagine base su cui poter eseguire le istruzioni successive. Viene eseguito un *pull* dell'immagine se non è stata ancora scaricata.

- `ENV <chiave> <valore> o <chiave>=<valore>`: imposta le variabili ambiente attribuendogli un identificativo ed un valore disponibili per l'utilizzo

in ogni istruzione che segue. La forma <chiave> <valore> permette di definire una sola variabile mentre la seconda forma permette di definirne di più con un unica istruzione.

- `RUN <comando> o ["comando","param1","..."]`: Esegue il comando specificato per poi utilizzare l'immagine ottenuta dal risultato dell'esecuzione per l'istruzione successiva
- `COPY <sorg.> <dest.>`: copia un *file* o una cartella all'interno del *filesystem* della macchina o del container nel percorso specificato (<nomecontainer>:<percorso> per indicare una sorgente e/o una destinazione all'interno del container)
- `ADD <sorg.> <dest.> o ["<src>",... "<dest>"]`: funzionamento analogo di COPY ma permette di utilizzare sorgenti URL e di includere più sorgenti
- `EXPOSE <porta>`: indica la porta alla quale il container ascolta all'esecuzione del codice, usando come default il protocollo TCP *(Transmission Control Protocol)* ma permettendo anche l'UDP *(User Datagram Protocol)*. Con EXPOSE la porta non viene esposta ma crea una documentazione tra chi crea l'immagine e chi esegue il container
- `ENTRYPOINT ["comando", "param1", "param2"]`: permette di eseguire un comando dopo la creazione del container, come se fosse un processo eseguibile.

Figura 2.4: Esempio di Dockerfile

```
#Immagine base
FROM mcr.microsoft.com/windows/servercore:ltsc2019

#Definizione variabili ambiente
ENV var1 = "" \
    var2 = "[]"

#Copia del file dalla macchina al container
COPY test.exe C:\

#Copia di più cartelle
ADD [Images, Data, Core, C:\]

#Esecuzione del codice passando il valore delle varaibili ambiente
RUN ["Core/build.exe", "$env:var1","$env:var2"]

#Esposizione sulla porta 80
EXPOSE 80

#Esecuzione in powershell dopo avvio del container
ENTRYPOINT ["powershell","test.exe"]
```

Una volta definita l'immagine base del container, ogni volta in cui viene specificato un comando, viene salvato lo stato del immagine su un nuovo strato, superiore allo strato precedente, che Docker riutilizza per velocizzare il processo di creazione dei container; inoltre le modifiche sono condivise tra le immagini, ottenendo un miglioramento di velocità, dimensione ed efficienza. Ogni volta che viene apportata una modifica, il registro delle modifiche offre il controllo totale sulle immagini "containerizzate" e delle loro versioni [9].
Uno dei maggiori vantaggi della stratificazione è la capacità di eseguire il *rollback* della versione di un' immagine nel caso ci fosse un errore o un calo di efficienza, in quel caso è possibile tornare alla versione precedente consentendo uno sviluppo agile, e garantendo CI/CD*(Continuous Integration/Continuous Delivery)*, ovvero un'automazione costante, monitorata continuamente in tutto il ciclo di vita dell'applicazione [4].

2.2 Kubernetes

Abbiamo appena visto come i container siano un ottimo modo per distribuire ed eseguire le applicazioni, in un ambiente di produzione è necessario garantire che non si verifichino interruzioni dei servizi, quindi se un container smette di funzionare c'è bisogno di un sistema che, in maniera automatica, si occupi di creare e avviare un nuovo container. Ed è proprio in questo frangente che entra in gioco *Kubernetes*, un *software open-source* reso disponibile da Google nel 2014 per l'automazione del Deployment, scalabilità, e gestione di applicazioni "containerizzate" in modo da raggrupparli in unità logiche per semplificare la gestione e la visibilità.
Kubernetes può esporre un container usando un nome DNS (Domain Name System) o il suo indirizzo IP, ed è grado di bilanciare il traffico di rete su più container se molto elevato, mantenendo il servizio stabile; permette di montare automaticamente un sistema di archiviazione a scelta (*storage locale*, dischi forniti da *cloud pubblici*) e di scegliere quante risorse come CPU e RAM ha bisogno ogni singolo container, Kubernetes allocherà i container adattando le loro risorse come richiesto.
Kubernetes permette inoltre di scegliere lo stato dei container utilizzati, ad esempio, per creare nuovi container, rimuovere quelli esistenti e adattando le risorse a quelle richieste dai nuovi, occupandosi anche di sostituire, terminale e bloccare il traffico a tutti i container che non rispondono, nei primi due casi, o che non possono ancora ricevere traffico, nell'ultimo caso.
Infine, Kubernetes consente di memorizzare e gestire informazioni sensibili, come le password e chiavi SSH *(Secure SHell)*, configurazione dell'applicazione e altre informazioni sensibili, dando la possibilità di aggiornale e distribuirle senza dover ricostruire le immagini dei container e senza svelarle nella configurazione del sistema [16].

Figura 2.5: Logo di Kubernetes

2.2.1 Introduzione al Cluster

Prima di procedere con la descrizione degli strumenti che permettono di gestire i container in maniera efficiente occorre parlare del sistema su cui risiedono. Il Cluster è un insieme di sistemi nodo che serve ad eseguire applicazioni "containerizzate", contenente un *nodo master*, che si occupa di mantenere lo stato desiderato del Cluster, decidendo le applicazioni da eseguire e le immagini di container utilizzate, e almeno un nodo di lavoro con il compito di eseguire le applicazioni e i carichi di lavoro contenente macchine di qualsiasi tipo(fisiche, virtuali, *cloud*, ecc.).
Per determinare quali applicazioni eseguire, le risorse utilizzabili e quali immagini scegliere per i container, il Cluster prevede uno stato desiderato definito da *file* di configurazione *manifest, file YAML o JSON* dove vengono indicati il tipo di applicazione da eseguire e il numero di repliche necessarie per eseguire un sistema, attraverso l'API *(Application Programming Interface)* Kubernetes, utilizzando il comando `kubectl` da riga di comando o usando l'API per interagire con il Cluster modificandone le impostazioni; sarà poi Kubernetes ad assicurasi che i container raggiungano lo stato desiderato, risolvendo gli errori che potrebbero presentarsi come arresti indesiderati di servizi, o errori nell'esecuzione di uno o più container [5].

2.2.2 Pod, Deployment e altri oggetti

Definito l'ambiente di sviluppo di Kubernetes, andiamo ad elencare tutti gli oggetti che rendono il *software* in questione, un ottimo orchestratore di servizi [3].

- Il primo oggetto da prendere in considerazione è il Pod, il più piccolo artefatto all'interno di un Cluster, rappresenta un insieme di container e volumi allocati sulla stessa macchina. Tutte le applicazioni di un Pod condividono lo stesso indirizzo IP, *hostname* e porte, e possono comunicare tramite coda di messaggi POSIX *(Portable Operating System Interface for Unix)* o IPC *(Inter-process Communication)*;

Figura 2.6: Esempio di Pod

```
apiVersion: v1
kind: Pod
metadata:
 name: esempiopod
spec:
 #Equivale a docker volume create
 volumes:
  - name: "volume-prova"
    hostPath:
     path: "C:/volume"
 #Docker run
 containers:
   - image: mcr.microsoft.com/windows/servercore
     name: win-prova
     #opzione -v
     volumeMounts:
       - mountPath: "C:/data"
         name: "win-volume"
     #opzione -p
     ports:
       - containerPort: 8080
         name: http
         protocol: TCP
```

- Quando vengono creati dei processi sul Cluster è di nostro interesse anche sapere dove trovarli per poter comunicare con loro, i Service Discovery hanno lo scopo di cercare con quale servizio e a quale indirizzo è in ascolto un processo dando una descrizione chiara sul tipo e usando un DNS per associare un nome ad uno o più Pod. Un Service Discovery di qualità è in grado di risolvere queste informazioni velocemente, con tempi di risposta brevi, aggiornamento immediato dei *client* non appena l'informazione associata al servizio cambia;

Figura 2.7: Esempio di Service

```
apiVersion: v1
kind: Service
metadata:
    name: esempio-service
spec:
  #Protocollo di default TCP
  ports:
    - name: http-port
      port: 80
      targetPort: 80
  selector:
    app: esempio-deploy
  type: LoadBalancer
```

- Se l'applicazione prevede il salvataggio o la creazione di dati, Kubernetes da la possibilità di creare un disco di archiviazione insieme al *filesystem* tramite l'uso dei volumi. Anche Docker prevede l'utilizzo di volumi ma c'è

differenza tra i due tipi, i volumi Docker sono una cartella su disco o su un altro container e non c'è nessuna gestione del loro ciclo di vita, cosa che in Kubernetes viene tenuta da conto comparandola al ciclo di vita del Pod, quando un Pod termina anche il volume cessa di funzionare. Anche se i container vengono riavviati tutti i dati salvati nel volume vengono preservati, ma la cosa che contraddistingue di più i volumi Kubernetes da quelli di Docker è che non solo un Pod può usare più volumi all'unisono ma permette l'uso tanti tipi di essi partendo da quelli utilizzati per infrastrutture specifiche come gli awsElasticBookStore se si utilizza Amazon Web Services o azureDisk per Microsoft Azure, fino a quelli *cross-platform* come i PersistentVolumeClaim i quali creano un archivio duraturo senza preoccuparsi dell'ambiente su cui è montato [21];

Figura 2.8: Esempio di Volume per Google Cloud Platform

```
#Oggetto utilizzato per inidicare il filesystem
#le zone dove vengono create le repliche del volume
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: esempio-storageclass
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  fsType: NTFS
  replication-type: regional-pd
  zones: us-central1-c, us-central1-a
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: esempio-volume
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 200Gi
  storageClassName: esempio-storageclass
```

- Per ora abbiamo parlato di oggetti che gestiscono processi a lungo periodo, è di nostro interesse anche definire quelli con cui poter utilizzare piccole esecuzioni di codice, come ad esempio il ripristino di un *database*, eseguite una volta sola. I Job creano ed eseguono Pod per poi cancellarli al termine dell'esecuzione del codice, in contrasto con i Pod tradizionali, provocando un ciclo infinito del codice se utilizzati al posto del Job. Se il processo non termina nel modo corretto il Job si occupa di sostituire il Pod con uno nuovo che esegue il processo da capo, finché non termina nel modo corretto;

Figura 2.9: Esempio di Job per esecuzione di una query su SQL Server

```
apiVersion: batch/v1
kind: Job
metadata:
  name: esempio-job
spec:
  template:
    spec:
      containers:
      - name: win-job
        image: microsoft/mssql-tools
        # -e su docker run
        env:
        - name: SA_PASSWORD
          value: "unapassword"
        command: ["/opt/mssql-tools/bin/sqlcmd"]
        args: ["-S", "server", "-U", "SA", "-P", "$(SA_PASSWORD)", "-Q", "SELECT * FROM TABELLA"]
      restartPolicy: Never
```

- I Pod sono oggetti con cui vengono eseguiti container con immagini che non cambiano nel tempo, i Deployment si occupano dell'aggiornamento alle nuove versioni delle immagini. Permettono inoltre di configurare il collaudo e le fasi progettuali(processo di *rollout*) dei Pod, con la possibilità di modificare il tempo tra un aggiornamento e l'altro dei Pod. Sono previsti anche dei controlli sulla stabilità delle nuove versioni (*health checks*) assicurandosi che le nuove applicazioni siano eseguite nella maniera corretta, in caso di un elevato numero di errori il Deployment viene fermato. Il *rollout* dei *software* è gestito dal *Deployment controller* eseguito all'interno del Cluster, in questo modo il Deployment può rimanere incustodito con la sicurezza che continuerà ad eseguire i container correttamente senza che si verifichi inattività o errori. Tutte queste funzionalità messe a disposizione, insieme alla possibilità di gestione lato *server* in zone con scarsa connessione internet, rendono i Deployment un ottimo strumento per il *rollout* di un immagine (Fig.2.10);

- Kubernetes possiede anche oggetti per archiviare e mettere in sicurezza i dati più sensibili come password e *security token*: i Secrets. Permettono di creare container facilmente trasferibili da un ambiente a un' altro dal fatto che derivano da immagini che non contengono dati sensibili;

Figura 2.10: Esempio di Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: esempio-deploy
spec:
  #Numero replice di container da creare
  replicas: 2
  selector:
    matchLabels:
      app: esempio-deploy
 template:
   metadata:
     labels:
       app: esempio-deploy
   spec:
     containers:
     - name: win-prova
       image: mcr.microsoft.com/windows/servercore
       ports:
       - containerPort: 80
       volumeMounts:
       - name: esempio-volume
         mountPath: "C:/volume"
     #Riferimento a un persistent volume esistente
     volumes:
     - name: esempio-volume
       persistentVolumeClaim:
         claimName: esempiovolume
```

2.2.3 Google Kubernetes Engine

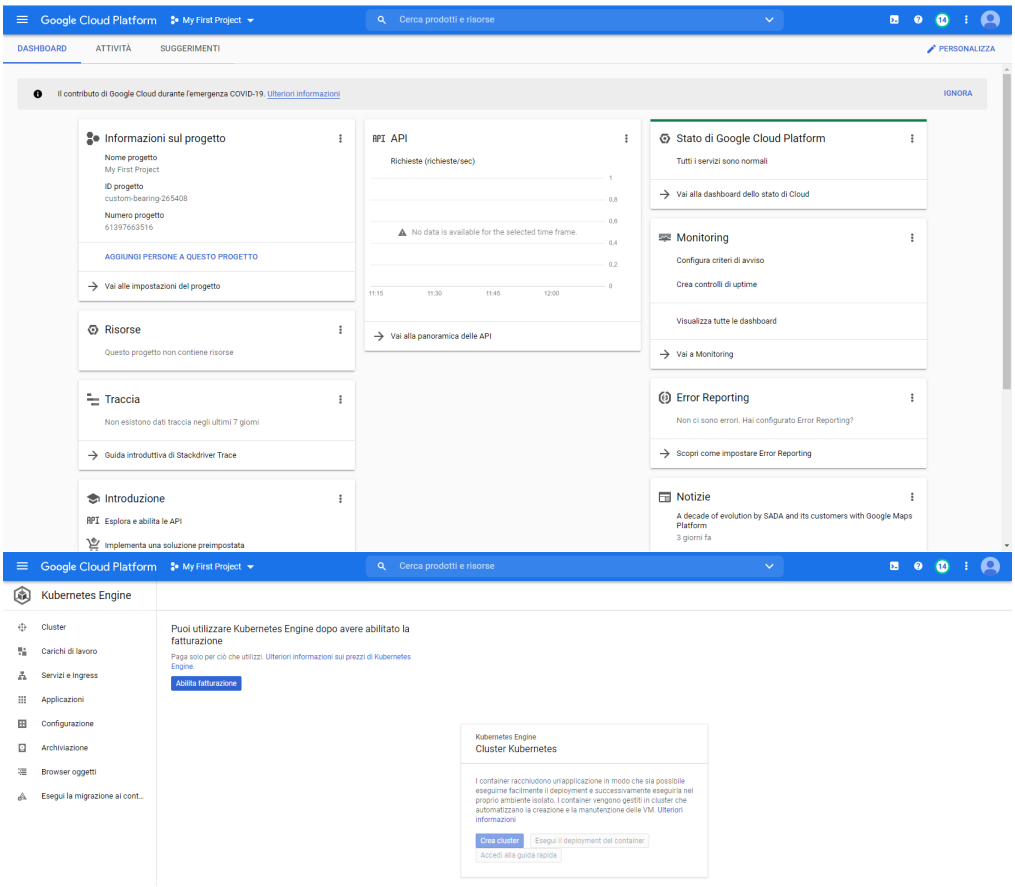
Ora che abbiamo una conoscenza approfondita degli strumenti principali di Kubernetes possiamo definire l'ambiente di lavoro che verrà utilizzato nel progetto di tesi. Google Cloud Platform (GCP) offre un insieme di servizi di *Cloud Computing* eseguiti sulla stessa infrastruttura che Google usa in Gmail, Youtube e altri servizi usati dagli utenti, fornendo modelli di servizio *Infrastructure as a service* (IaaS), *Platform as a service* (PaaS), *Function as a Service* (FaaS) e *Container as a Service* (CaaS) tramite Google Kubernetes Engine, il servizio utilizzato [11].

Figura 2.11: Logo di Google Cloud Platform e GKE

Google Kubernetes Engine (GKE) mette a disposizione un ambiente di sviluppo per applicazioni "containerizzate" attraverso l'infrastruttuta Google, permettendo di scegliere Cluster sempre aggiornati e disponibili grazie ai *Site Reliability Engineer* [13]. Gli utenti possono personalizzare i Cluster in base alla stabilità

della versione, ai requisiti di traffico tra i Pod e al suo isolamento, controllato attraverso i canali di rilascio dando la possibilità di scegliere i tipi di aggiornamento (rapidi, regolari o stabili), e possono controllare l'accesso tramite account Google, definendo autorizzazioni opportune in base ai ruoli. È supportato il formato standard delle immagini Docker per carichi di lavoro Linux e Windows eseguendo nodi Windows Server (fino alla versione 1909) e Linux, permettendo di archiviare e accedere alle immagini attraverso il Container Registry automaticamente.
Con la riparazione automatica, GKE avvia un processo di riparazione per un nodo che non supera il controllo di integrità, consente di specificare la quantità di CPU e memoria (RAM) di cui ha bisogno ciascun container, permettendo, inoltre, di creare e collegare un archiviazione permanente in formato HDD o SSD ai container e ospitare *database* completi; le unità SSD consentono un elevato tasso di operazioni di *input/output* al secondo con una latenza bassa rispetto ai dischi permanenti essendo collegate fisicamente al server ospitante delle macchine virtuali. I carichi di lavoro vengono distribuiti tra i diversi pool di istanze in diverse aree geografiche, per garantire i massimi livelli di prestazioni, velocità effettiva e disponibilità, con costi ridotti e la possibilità di esportarli in altre piattaforme Kubernetes nei *cloud* e *on-premise*.
GKE fornisce una visibilità sui Cluster fornendo informazioni come l'utilizzo delle risorse suddiviso in base a *namepace* ed etichette, e i costi dell'utilizzo delle risorse, pagandole mensilmente in base al tempo di calcolo utilizzato, ai nuovi utenti viene fornito un *voucher* valido per un anno per iniziare ad utilizzare questo servizio [12].

Figura 2.12: Interfaccia di GCP e GKE



Capitolo 3

ERP e Bike

Definita la struttura e gli strumenti utilizzati, prima di procedere con l'implementazione dell'applicazione è necessario comprendere la struttura e il funzionamento di quest'ultima partendo dai concetti generali di un sistema ERP per poi descrivere nel dettaglio il *software* Bike.

3.1 ERP

Come già accennato nella parte introduttiva, un *Enterprise Resource Planning* (ERP) è un *software* fondamentale per aziende di qualunque grandezza e tipo in quanto permettono di gestire attività commerciali quali contabilità, *project management* e gestione del rischio, in grado di pianificare, quantificare, prevedere e comunicare i risultati finanziari. Un sistema ERP raccoglie tutti dati transazionali provenienti da diverse fonti dell'organizzazione in un'unica fonte affidabile senza incorrere nel rischio di creazione di duplicati, e definisce un insieme di processi di *business* per garantire lo scambio di quest'ultimi [23].

3.1.1 Struttura

Il principio che sta alla base di un sistema ERP è la raccolta centralizzata dei dati di ogni reparto dell'azienda, per questo motivo la struttura dati si basa sulla condivisione di un *database* dove i dati di ogni reparto vengono raccolti organizzati e aggiornati costantemente (Fig.3.1), assicurandosi che vengano visualizzati agli utenti giusti per ogni settore interno.
Perciò in alternativa all'utilizzo di diversi *database standalone* contenenti un vasto numero di fogli di calcolo tra loro scollegati, i sistemi ERP consentono a tutti gli utenti, di creare, archiviare e usare gli stessi dati derivanti da processi comuni. Con un *repository* di dati sicuro e centralizzato, tutti all'interno dell'organizzazione possono essere certi che i dati siano corretti, aggiornati e completi. L'integrità dei dati è garantita per ogni attività eseguita all'interno dell'organizzazione, dalle

dichiarazioni finanziarie trimestrali al singolo *report* dei crediti, senza usare fogli di calcolo soggetti a errore.

Figura 3.1: Struttura di un sistema ERP

L'architettura utilizzata solitamente è quella *client-server*, viene creata una rete dove tutti i *client* dell'azienda si collegano e fanno confluire le loro informazioni su un unico server, che interfaccia il *database management systems*. Il sistema ERP è quindi un sistema modulare e aperto dove ogni modulo rappresenta una funzione specifica dell' azienda personalizzabile e selezionabile in base alla necessità dell'impresa, e deve essere anche in grado di interfacciarsi con sistemi diversi ed eterogenei tra di loro [23].

3.1.2 Vantaggi nell'utilizzo

Tutte le caratteristiche della struttura di un ERP comportano una serie di vantaggi per l'azienda, primi fra questi, minori costi di gestione e costi operativi tramite sistemi uniformi e integrati e processi business semplificati. Porta a una maggior collaborazione tra gli utenti che condividono i dati, evitando, oltre alla duplicazione, eventuali incomprensioni sulla descrizione e interpretazione di essi, per richieste e ordini di acquisto. Le informazioni sono aggiornate in tempo reale e può integrarsi con altri sistemi, rendendo possibile il dialogo con programmi preesistenti senza cambiare nessun dato [20].

3.2 Bike

Bike è un *software* ERP sviluppato da Meta Informatica S.R.L. [17] utilizzato dalla maggior parte delle aziende nelle provincie di Ascoli Piceno, Macerata e Pesaro Urbino. Bike permette di controllare e gestire in qualsiasi piattaforma, tutte le aree funzionali, le attività amministrative delle aziende di dimensioni e settori diversi in sicurezza e stabilità , consentendo, inoltre, di reperire correlare e rielaborare rapidamente le informazioni, dando una visione completa di tutti gli aspetti contabili. È possibile gestire più aziende contemporaneamente, lavorare in rete con più utenti e comunicare in maniera completa con i principali *software* garantendo la migrazione dei dati e il trasferimento di qualsiasi movimento [2].

Figura 3.2: Logo di Meta Informatica SRL e del software Bike

Bike è stato sviluppato utilizzando Instant Developer [14], un *Integrated Development Environment* (IDE) che permette lo sviluppo di applicazioni multipiattaforma e multicanale che utilizzano tecnologie Java e .NET, utilizzando un solo modello applicativo ed una sola base di codice per *app web*, *mobile* e per qualunque altro canale e permettendo di modificare *file* di compilazione creati da altri IDE. Permette di creare applicazioni in modo semplice e veloce, riducendo in modo drastico i costi di sviluppo e quelli di manutenzione grazie alla "programmazione relazionale" una declinazione della programmazione *model driven*, in questo modo l'intero progetto viene memorizzato in un grafo di relazioni (nodi di rete) e dipendenze invece che in *file* di testo, consentendo di modificare un punto qualunque e tutto il resto del sistema si adatterà automaticamente [15].
Instant Developer offre una libreria Javascript attraverso il *Framework RD3* [18] la quale realizza il funzionamento direttamente nel *browser* e invia gli eventi avvenuti al server web, tramite una comunicazione XML dello stato dell'interfaccia dell'applicazione alla libreria. RD3 garantisce l aumento del livello di interattività delle applicazioni web offrendo la stessa esperienza utente delle applicazioni *client*, utilizza meno banda passante e risorse di elaborazione lato server e rende asincrona la comunicazione fra *browser* e *server* per creare modelli di interfaccia utente animate e responsive [18].

Figura 3.3: Logo di Instant Developer

3.2.1 Base di dati

La base di dati di Bike creata utilizzando Instant Devolper e resa attiva tramite il servizio di Microsoft SQL Server [19] è composta da 4 *database*, 1 contiene i dai d'accesso e i permessi di ogni utente, 1 è utilizzato per la comunicazione interna e per attività di supporto, l'ultimo corrisponde al *database* della struttura ERP contenente tutti i dati di ogni reparto dell'azienda:

- AMBIENTI: Contiene le tabelle con i dati di accesso degli utenti, le email dell'azienda per area funzionale e i permessi degli utenti in base al loro ruolo;
- BIKE031: è il *database* condiviso della struttura ERP, contenente tutti i dati di ogni reparto dell'azienda che gli utenti posso visualizzare come ad esempio le fatture, l'elenco dei notai, l'inventario del magazzino, i fornitori e gli ordini;
- CHARLY031: Contiene tutti i dati per gestire funzioni quali ricerca del CAP, Codice Fiscale e altri strumenti di supporto;
- BIKE031_ALLEGATI: Contiene gli allegati caricati e inseriti nelle email;

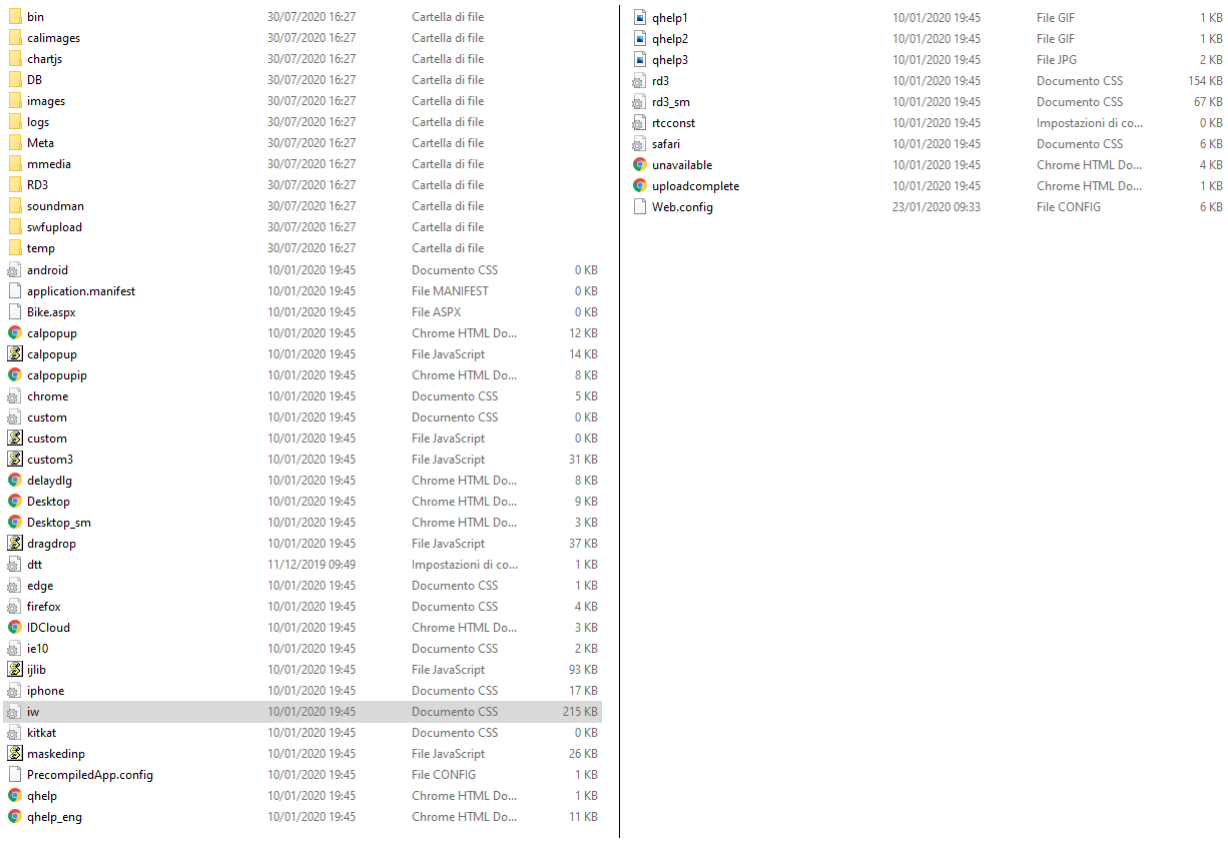
3.2.2 Applicazione Web

Alla compilazione del codice Instant Developer crea tutti i *file* e le directory dell'applicazione Bike in modo tale da esse compatibile, con il *Framework ASP.NET* a 32 bit, utilizzandolo insieme a RD3. Basato sulla piattaforma di sviluppo .NET usufruisce dei suoi componenti base come librerie per lavorare con stringhe, *file* e altri dati con codice C#, F#, Visual Basic e aggiungendo ulteriori librerie e funzionalità per costruire applicazioni specifiche. ASP.NET è in grado di processare richiese web in C# or F#, permette di creare pagine web dinamiche in C# tramite una sintassi di *Web-page templating syntax*. Le librerie aggiunte permettono di utilizzare pattern architetturali come quello MVC (Model View Controller) e

permettono l'autenticazione a multi-fattori ed esterna tramite Google, Facebook, Twitter ecc grazie al sistema di autenticazione fornito dal *Framework*. Include inoltre estensioni editor per completamento del codice, e altre funzionalità per lo sviluppo di pagine web [1].
Nel caso di Bike la struttura creata è la seguente:

- bin: Risiedono i *file assembly* compilati per i componenti, controlli o altro codice da usare all'interno dell'applicazione. Ad ogni nuova compilazione del sito vengono creati nuovi *file .dll*.
- images e calimages : Contengono tutte le immagini utilizzate nella UI *(User Interface)*;
- Meta: Directory personalizzata contenente il *file* per la connessione alla base di dati, e un serie di cartelle dove sono salvate le firme delle e-mail, le impostazioni di visualizzazione del Menu per ogni singolo utente, *template* per stampare fatture e altra documentazione e un *file .pdf* contenente tutte le release fino all'ultima versione
- logs: Vengono salvati i *file log* e il registro degli errori riscontrati ordinati per data e ora, è possibile disabilitarli per migliorare i tempi di risposta dell'applicazione senza appesantirla troppo;
- RD3: contiene le impostazioni in codice JavaScript basate su *Framework RD3* per implementare pagine web dinamiche, la comunicazione XML e altre funzionalità fornite dal *Framework*
- temp: Archivio dei *file* temporanei
- File .css edge,firefox,chrome,safari,android,iphone: Definiscono le regole di formattazione a senconda del tipo di *browser* o dispositivo usato nel caso di android.css e iphone.css
- Bike.aspx: Corrisponde al fine home.html, qui viene caricata ogni pagina da visualizzare;
- Web.config: Permette di configurare il tipo di *debug* e quali errori visualizzare, il tipo di autenticazione e gli utenti autorizzati ad accedere. Permette di scegliere il nome del *file .aspx* dove verranno caricati gli elementi della pagina;

Figura 3.4: Framework di Bike



Capitolo 4

Sviluppo dei container e implementazione in GKE

4.1 Progettazione

Definito anche il funzionamento di Bike si passa alla "containerizzazione" vera e propria del software Bike tramite container. Inizieremo col definire i requisiti dell'applicazione, in particolar modo, le immagini Docker da utilizzare con relative impostazioni, seguito dal caricamento delle immagini sul Container Registry e loro implementazione su Kubernetes tramite GKE, una volta verificato il corretto funzionamento, renderemo accessibile l'applicazione a tutti gli utenti.

4.1.1 Analisi dei requisiti

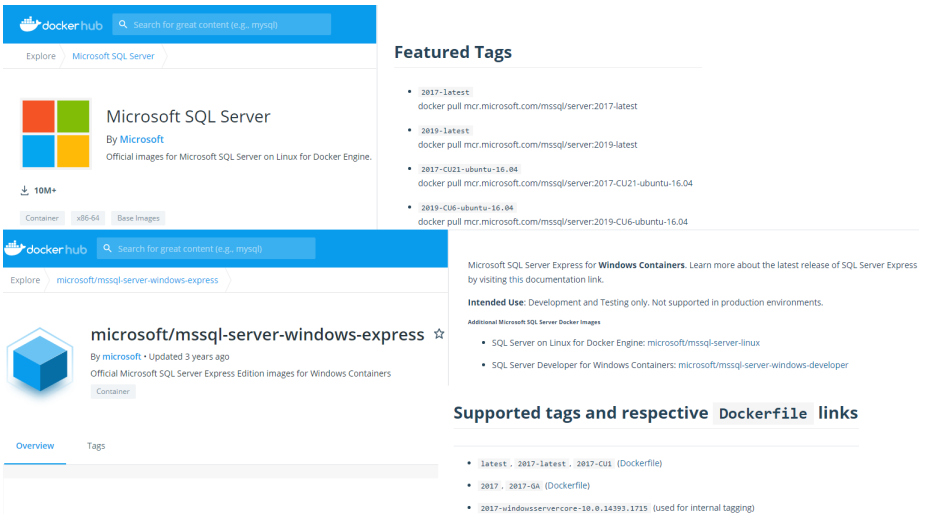
Come abbiamo già detto, lo scopo del nostro progetto è quello di "containerizzare" un applicativo web e gestire, attraverso un servizio di *Cloud Computing*, le sue richieste con un orchestratore. Dobbiamo allora definire in prima battuta quali immagini utilizzare per creare i container che permettono il corretto funzionamento del *software*. Dato che Bike utilizza il *Framework* ASP.NET per l'applicazione web e Microsoft SQL Server per la base di dati utilizzeremo un immagine ASP.NET Framework 4.8, e un immagine SQL Server per la base di dati per poi eseguire il backup dei dati dai *file .mdf e .ldf* inseriti all'interno del container.

4.1.2 Creazione delle immagini

La prima immagine che andremo a creare è quella della base di dati; Docker Hub mette a disposizione immagini SQL Server su Kernel Linux e Windows. Utilizzando l'immagine Windows è possibile scaricare dal *repository* GitHub e modificare il Dockerfile della versione Developer o Express. Il funzionamento dell'immagine analogo a seconda del Kernel su cui si appoggia, dando piena libertà di scelta

a seconda della situazione. Procederemo prima con la creazione dell'immagine SQL Server su Windows e a seguire la creazione della corrispondente in Linux.

Figura 4.1: Archivio Immagine SQL Server Windows e Linux



Per ripristinare la base di dati sono necessari *file backup* di SQL Server; per ogni *database* sono presenti un *file .mdf* ed un *file .ldf*, oppure un *file .bak* che dovranno essere spostati all'interno del container. Perciò andremo a modificare il Dockerfile fornito aggiungendo un istruzione di copia, in questo caso `ADD`, per spostare la cartella dove sono inseriti i suddetti *file* insieme all'istruzione `EXPOSE` per consentire la comunicazione tra base di dati e applicazione.

Figura 4.2: Immagine base di dati Windows

```
FROM mcr.microsoft.com/windows/servercore:1909

LABEL maintainer "Perry Skountrianos"

# Download Links:
ENV sql_express_download_url "https://go.microsoft.com/fwlink/?linkid=829176"

#Variabili ambienti per parametri base dell'immagine
ENV sa_password="_" \
    attach_dbs="[]" \
    ACCEPT_EULA="_" \
    sa_password_path="C:\ProgramData\Docker\secrets\sa-password"

SHELL ["powershell", "-Command", "$ErrorActionPreference = 'Stop'; $ProgressPreference = 'SilentlyContinue';"]

# make install files accessible
COPY start.ps1 /
WORKDIR /

#Installzazione del Server SQL
RUN Invoke-WebRequest -Uri $env:sql_express_download_url -OutFile sqlexpress.exe ; \
        Start-Process -Wait -FilePath .\sqlexpress.exe -ArgumentList /qs, /x:setup ; \
        .\setup\setup.exe /q /ACTION=Install /INSTANCENAME=SQLEXPRESS /FEATURES=SQLEngine /UPDATEENABLED=0 /SQLSVCACCOUNT='NT AUTHORITY\System' |
        /SQLSYSADMINACCOUNTS='BUILTIN\ADMINISTRATORS' /TCPENABLED=1 /NPENABLED=0 /IACCEPTSQLSERVERLICENSETERMS ; \
        Remove-Item -Recurse -Force sqlexpress.exe, setup

#Arresto del servizio per il salvataggio delle impostazioni
RUN stop-service MSSQL`$SQLEXPRESS ; \
        set-itemproperty -path 'HKLM:\software\microsoft\microsoft sql server\mssql14.SQLEXPRESS\mssqlserver\supersocketnetlib\tcp\ipall' -name tcpdynamicports -value '' ; \
        set-itemproperty -path 'HKLM:\software\microsoft\microsoft sql server\mssql14.SQLEXPRESS\mssqlserver\supersocketnetlib\tcp\ipall' -name tcpport -value 1433 ; \
        set-itemproperty -path 'HKLM:\software\microsoft\microsoft sql server\mssql14.SQLEXPRESS\mssqlserver\' -name LoginMode -value 2 ;

#Avvio dell'applicazione con paramentri
CMD .\start -sa_password $env:sa_password -ACCEPT_EULA $env:ACCEPT_EULA -attach_dbs \"$env:attach_dbs\" -Verbose

#Copia dei file di backup per ripristino della base di dati
ADD META C:\\META

EXPOSE 1433
```

A questo punto possiamo eseguire il comando `docker build` per avviare il processo di creazione immagine attribuendoli un nome con l'opzione `-t`; durante il processo di creazione si possono osservare tutti gli strati creati da Docker per ogni istruzione eseguita.

Figura 4.3: Creazione immagine base di dati Windows

Prima di avviare il container è necessario un volume per salvare i dati della base di dati, per contenere tutti i dati della base di dati che verranno successivamente ripristinati. Il volume viene creato tramite il comando `docker volume create`. Con il comando `docker run` viene creato il container dell'immagine appena definita, assegnandoli un identificativo e un nome. È possibile, inoltre, specificare la porta di destinazione del container, le variabili ambiente da passare e il volume da assegnare al container, tramite una serie di opzioni previste dallo stesso comando.

Figura 4.4: Creazione container server Windows

```
C:\WINDOWS\system32>docker volume create volume-bike
volume-bike
C:\WINDOWS\system32>docker run -e ACCEPT_EULA="Y" -e SA_PASSWORD=Password-Bike2019 -p 1433:1433 -v volume-bike:C:\Bike -d bike-server:windows
a42de042ff25ad77fcb8c177b04e6d5b1072ac769f9834e556317a2338d18b63

C:\WINDOWS\system32>docker container ls
CONTAINER ID        IMAGE                 COMMAND                  CREATED             STATUS              PORTS                    NAMES
a42de042ff25        bike-server:windows   "powershell -Command…"   7 minutes ago       Up 7 minutes        0.0.0.0:1433->1433/tcp   affectionate_beaver
```

Una volta che il container sarà operativo, verrà eseguito il comando per il ripristino dei *database* tramite `sqlcmd` utilizzando in coppia i *file .mdf e .ldf* per ogni *database* da creare tramite il comando `docker exec`.

Figura 4.5: Comando di ripristino base di dati

```
C:\WINDOWS\system32>docker exec -it a42de042ff25 sqlcmd -U SA -P Password-Bike2019 -Q "CREATE DATABASE [AMBIENTI] ON (FILENAME = 'C:\META\BIKE\DATI\AMBIENTI_Data.mdf'),(FILENAME = 'C:\META\BIKE\DATI\AMBIENTI
_Log.ldf') FOR ATTACH"
C:\WINDOWS\system32>docker exec -it a42de042ff25 sqlcmd -U SA -P Password-Bike2019 -Q "CREATE DATABASE [BIKE031] ON (FILENAME = 'C:\META\BIKE\DATI\BIKE_Data.mdf'),(FILENAME = 'C:\META\BIKE\DATI\BIKE_Log.ldf'
) FOR ATTACH"
C:\WINDOWS\system32>docker exec -it a42de042ff25 sqlcmd -U SA -P Password-Bike2019 -Q "CREATE DATABASE [CHARLY031] ON (FILENAME = 'C:\META\BIKE\DATI\CHARLY_Data.mdf'),(FILENAME = 'C:\META\BIKE\DATI\CHARLY_Lo
g.ldf') FOR ATTACH"
C:\WINDOWS\system32>docker exec -it a42de042ff25 sqlcmd -U SA -P Password-Bike2019 -Q "CREATE DATABASE [BIKE031_ALLEGATI] ON (FILENAME = 'C:\META\BIKE\DATI\BIKE031_ALLEGATI_Data.mdf'),(FILENAME = 'C:\META\BI
KE\DATI\BIKE031_ALLEGATI_Log.ldf') FOR ATTACH"
```

Il procedimento per la creazione dell' immagine Linux è analogo a quello appena visto; quello che cambia è la struttura del Dockerfile, più contenuta rispetto all'immagine Windows e contenente solo le istruzioni essenziali per la modifica dell'immagine originaria.

Figura 4.6: Immagine base di dati Linux

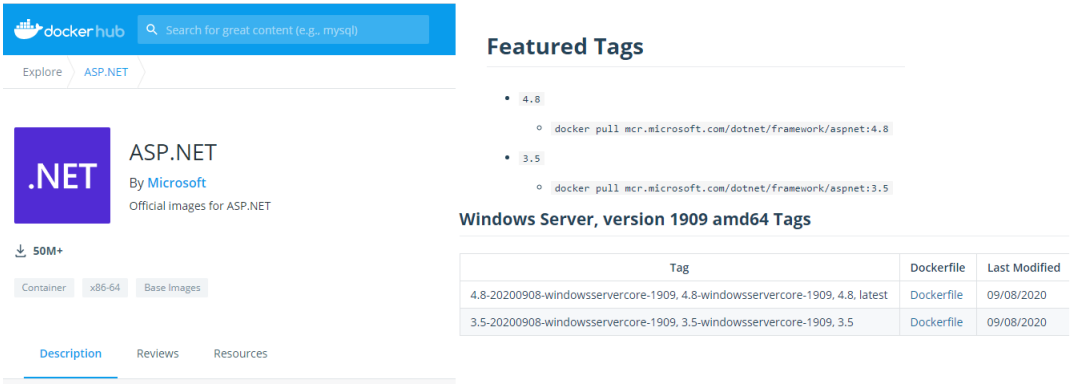
```
FROM microsoft/mssql-server-linux:latest

#Creazione cartella META
RUN mkdir /var/opt/META

#Copia dei file di backup per ripristino della base di dati
COPY META /var/opt/META

EXPOSE 1433
```

Figura 4.7: Creazione immagine base di dati e container Linux con ripristino

```
>docker build -t bike-server:linux C:\                                    \Build_immagine_server\linux
Sending build context to Docker daemon  1.314GB
Step 1/4 : FROM microsoft/mssql-server-linux:latest
 ---> 314918ddaedf
Step 2/4 : RUN mkdir /var/opt/META
 ---> Running in 3487dc9adbff
Removing intermediate container 3487dc9adbff
 ---> 430b919876c9
Step 3/4 : COPY META /var/opt/META
 ---> 2ef3d5dc08cf
Step 4/4 : EXPOSE 1433
 ---> Running in abd27e54e94d
Removing intermediate container abd27e54e94d
 ---> cd47e3ae9b02
Successfully built cd47e3ae9b02
Successfully tagged bike-server:linux
C:\WINDOWS\system32>docker run -e ACCEPT_EULA="Y" -e SA_PASSWORD=Password-Bike2019 -p 1433:1433 -v volume-bike:\Bike -d bike-server:linux
355adc12ccd1082a125bd92d9b7d2af10fb657e14f21caafe60d5d952e60a24f

C:\WINDOWS\system32>docker container ls
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
355adc12ccd1        bike-server:linux   "/opt/mssql/bin/sqls…"   3 minutes ago       Up 3 minutes        0.0.0.0:1433->1433/tcp   bold_wright
C:\WINDOWS\system32>docker exec -it 355adc12ccd1 /opt/mssql-tools/bin/sqlcmd -U SA -P Password-Bike2019 -Q "CREATE DATABASE [AMBIENTI] ON (FILENAME = '/var/opt/META/BIKE/DATI/AMBIENTI_Data.mdf'),(FILENAME =
'/var/opt/META/BIKE/DATI/AMBIENTI_Log.ldf') FOR ATTACH"

C:\WINDOWS\system32>docker exec -it 355adc12ccd1 /opt/mssql-tools/bin/sqlcmd -U SA -P Password-Bike2019 -Q "CREATE DATABASE [BIKE031] ON (FILENAME = '/var/opt/META/BIKE/DATI/BIKE_Data.mdf'),(FILENAME = '/var
/opt/META/BIKE/DATI/BIKE_Log.ldf') FOR ATTACH"

C:\WINDOWS\system32>docker exec -it 355adc12ccd1 /opt/mssql-tools/bin/sqlcmd -U SA -P Password-Bike2019 -Q "CREATE DATABASE [CHARLY031] ON (FILENAME = '/var/opt/META/BIKE/DATI/CHARLY_Data.mdf'),(FILENAME = '
/var/opt/META/BIKE/DATI/CHARLY_Log.ldf') FOR ATTACH"

C:\WINDOWS\system32>docker exec -it 355adc12ccd1 /opt/mssql-tools/bin/sqlcmd -U SA -P Password-Bike2019 -Q "CREATE DATABASE [BIKE031_ALLEGATI] ON (FILENAME = '/var/opt/META/BIKE/DATI/BIKE031_ALLEGATI_Data.md
f'),(FILENAME = '/var/opt/META/BIKE/DATI/BIKE031_ALLEGATI_Log.ldf') FOR ATTACH"
```

Una volta creato il *database* possiamo passare all'applicazione web. Anche in questo caso DockerHub mette a disposizione un'immagine per applicazioni che lavorano con il *Framework ASP.NET* con relativo Dockerfile da scaricare e modificare.

Figura 4.8: Archivio immagine ASP.NET



Esattamente come nel caso della creazione della base di dati, il Dockerfile viene modificato aggiungendo, oltre all' `EXPORT` nella porta 80 anche il comando di copia dell'applicazione web nella cartella "wwwroot" all'interno di "C:\inetpub". Se proviamo a costruire l'immagine e creare un container l'applicazione di Bike darà errore, questo perché Bike funziona a 32 bit e l'immagine funziona a 64 bit con l'opzione per l'esecuzione di applicazioni a 32 bit disabilitata, perciò dovremmo inserire all'interno del Dockerfile un ulteriore comando per abilitazione dell'opzione (Fig.4.10). Prima di creare l'immagine dobbiamo assicurarci che il *file ini.txt* all'interno della cartella Meta dell' applicazione, sia impostato il nome del container SQL Server per consentire l'interrogazione della base di dati.

Figura 4.9: File ini.txt

```
1:SQLOLEDB:<nome container server>:AMBIENTI:sa:<password sa>
1:SQLOLEDB:<nome container server>:BIKE031:sa:<password sa>
1:SQLOLEDB:<nome container server>:CHARLY031:sa:<password sa>
1:SQLOLEDB:<nome container server>:BIKE031_ALLEGATI:sa:<password sa>
```

Figura 4.10: Immagine ASP.NET per Bike

```
# escape=`

ARG REPO=mcr.microsoft.com/dotnet/framework/runtime
FROM $REPO:4.8-windowsservercore-1909

SHELL ["powershell", "-Command", "$ErrorActionPreference = 'Stop'; $ProgressPreference = 'SilentlyContinue';"]

#Abilitazione del Framework, IIS e Web Server
RUN Add-WindowsFeature Web-Server; `
    Add-WindowsFeature NET-Framework-45-ASPNET; `
    Add-WindowsFeature Web-Asp-Net45; `
    Remove-Item -Recurse C:\inetpub\wwwroot\*; `
    Invoke-WebRequest -Uri https://dotnetbinaries.blob.core.windows.net/servicemonitor/2.0.1.10/ServiceMonitor.exe -OutFile C:\ServiceMonitor.exe

# Installazione del compilarore Roslyn e generatore di immagini native
RUN Invoke-WebRequest https://api.nuget.org/packages/microsoft.net.compilers.2.9.0.nupkg -OutFile c:\microsoft.net.compilers.2.9.0.zip; `
    Expand-Archive -Path c:\microsoft.net.compilers.2.9.0.zip -DestinationPath c:\RoslynCompilers; `
    Remove-Item c:\microsoft.net.compilers.2.9.0.zip -Force; `
    &C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ngen.exe update; `
    &C:\Windows\Microsoft.NET\Framework\v4.0.30319\ngen.exe update; `
    &C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ngen.exe install c:\RoslynCompilers\tools\csc.exe /ExeConfig:c:\RoslynCompilers\tools\csc.exe | `
    &C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ngen.exe install c:\RoslynCompilers\tools\vbc.exe /ExeConfig:c:\RoslynCompilers\tools\vbc.exe | `
    &C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ngen.exe install c:\RoslynCompilers\tools\VBCSCompiler.exe /ExeConfig:c:\RoslynCompilers\tools\VBCSCompiler.exe |
    &C:\Windows\Microsoft.NET\Framework\v4.0.30319\ngen.exe install c:\RoslynCompilers\tools\csc.exe /ExeConfig:c:\RoslynCompilers\tools\csc.exe | `
    &C:\Windows\Microsoft.NET\Framework\v4.0.30319\ngen.exe install c:\RoslynCompilers\tools\vbc.exe /ExeConfig:c:\RoslynCompilers\tools\vbc.exe | `
    &C:\Windows\Microsoft.NET\Framework\v4.0.30319\ngen.exe install c:\RoslynCompilers\tools\VBCSCompiler.exe  /ExeConfig:c:\RoslynCompilers\tools\VBCSCompiler.exe

ENV ROSLYN_COMPILER_LOCATION c:\RoslynCompilers\tools

EXPOSE 80

#Enable 32bit application
RUN cmd.exe --% /c %windir%\system32\inetsrv\appcmd set config -section:applicationPools -applicationPoolDefaults.enable32BitAppOnWin64:true

#Inserimento dei file dell'applicazione all'interno del container
ADD Bike C:\\inetpub\\wwwroot

ENTRYPOINT ["C:\\ServiceMonitor.exe", "w3svc"]
```

Figura 4.11: Creazione immagine ASP.NET per Bike

```
>docker build -t bike-web C:\                                                \Build_immagine_web_funzionante:official
Sending build context to Docker daemon  43.57MB
Step 1/10 : ARG REPO=mcr.microsoft.com/dotnet/framework/runtime
Step 2/10 : FROM $REPO:4.8-windowsservercore-1909
 ---> abe9de730745
Step 3/10 : SHELL ["powershell", "-Command", "$ErrorActionPreference = 'Stop'; $ProgressPreference = 'SilentlyContinue';"]
 ---> Using cache
 ---> 9cdff9d0b524
Step 4/10 : RUN Add-WindowsFeature Web-Server;     Add-WindowsFeature NET-Framework-45-ASPNET;     Add-WindowsFeature Web-Asp-Net45;     Remove-Item -Recurse C:\inetpub\wwwroot\*;     Invoke-WebRequest -Uri
https://dotnetbinaries.blob.core.windows.net/servicemonitor/2.0.1.10/ServiceMonitor.exe -OutFile C:\ServiceMonitor.exe
 ---> Running in 8b53e931cfc0

Success Restart Needed Exit Code      Feature Result
------- -------------- ---------      --------------
True    No             Success        {Common HTTP Features, Default Documen...
WARNING: Failed to start automatic updating for installed components. Error:
0x80080005
True    No             Success        {ASP.NET 4.8}
WARNING: Failed to start automatic updating for installed components. Error:
0x80080005
True    No             Success        {Application Development, ASP.NET 4.8,...
WARNING: Failed to start automatic updating for installed components. Error:
0x80080005


Removing intermediate container 8b53e931cfc0
 ---> 21cefe395c24
Step 5/10 : RUN Invoke-WebRequest https://api.nuget.org/packages/microsoft.net.compilers.2.9.0.nupkg -OutFile c:\microsoft.net.compilers.2.9.0.zip;     Expand-Archive -Path c:\microsoft.net.compilers.2.9.0.z
ip -DestinationPath c:\RoslynCompilers;     Remove-Item c:\microsoft.net.compilers.2.9.0.zip -Force;     &C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ngen.exe update;     &C:\Windows\Microsoft.NET\Framew
ork\v4.0.30319\ngen.exe update;     &C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ngen.exe install c:\RoslynCompilers\tools\csc.exe /ExeConfig:c:\RoslynCompilers\tools\csc.exe |     &C:\Windows\Microsoft.
NET\Framework64\v4.0.30319\ngen.exe install c:\RoslynCompilers\tools\vbc.exe /ExeConfig:c:\RoslynCompilers\tools\vbc.exe |     &C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ngen.exe install c:\RoslynCompi
lers\tools\VBCSCompiler.exe /ExeConfig:c:\RoslynCompilers\tools\VBCSCompiler.exe |     &C:\Windows\Microsoft.NET\Framework\v4.0.30319\ngen.exe install c:\RoslynCompilers\tools\csc.exe /ExeConfig:c:\RoslynCom
pilers\tools\csc.exe |     &C:\Windows\Microsoft.NET\Framework\v4.0.30319\ngen.exe install c:\RoslynCompilers\tools\vbc.exe /ExeConfig:c:\RoslynCompilers\tools\vbc.exe |     &C:\Windows\Microsoft.NET\Framewo
rk\v4.0.30319\ngen.exe install c:\RoslynCompilers\tools\VBCSCompiler.exe  /ExeConfig:c:\RoslynCompilers\tools\VBCSCompiler.exe
 ---> Running in cfabb6e3c21b3
Microsoft (R) CLR Native Image Generator - Version 4.8.3752.0
Copyright (c) Microsoft Corporation.  All rights reserved.
1>    Compiling assembly c:\RoslynCompilers\tools\VBCSCompiler.exe (CLR v4.0.30319) ...
Removing intermediate container cfabb6e3c21b3
 ---> 3f28d581f039
Step 6/10 : ENV ROSLYN_COMPILER_LOCATION c:\RoslynCompilers\tools
 ---> Running in 79fff3709689
Removing intermediate container 79fff3709689
 ---> cb03d0c8fb28
Step 7/10 : EXPOSE 80
 ---> Running in 66c841cc3746
Removing intermediate container 66c841cc3746
 ---> 5f5c6ff14ba1
Step 8/10 : RUN cmd.exe --% /c %windir%\system32\inetsrv\appcmd set config -section:applicationPools -applicationPoolDefaults.enable32BitAppOnWin64:true
 ---> Running in 8ebc4ce011b4
Applied configuration changes to section "system.applicationHost/applicationPools" for "MACHINE/WEBROOT/APPHOST" at configuration commit path "MACHINE/WEBROOT/APPHOST"
Removing intermediate container 8ebc4ce011b4
 ---> d438728fe32a
Step 9/10 : ADD Bike C:\\inetpub\\wwwroot
 ---> cefd3ff4903d
Step 10/10 : ENTRYPOINT ["C:\\ServiceMonitor.exe", "w3svc"]
 ---> Running in e45281c2435f
Removing intermediate container e45281c2435f
 ---> c7835b95d73c
Successfully built c7835b95d73c
Successfully tagged bike-web:official
```

4.2 Implementazione

4.2.1 Creazione del Cluster e Container Registry

Accertato il corretto funzionamento dei container creati possiamo usare le immagini per creare container tramite Kubernetes utilizzando Google Kubernetes Engine. Prima di procedere con la creazione del Cluster dobbiamo configurare Docker in modo tale da permettere il caricamento delle immagini sul Container Registry di Google Cloud. Una volta attivato il servizio di Google Cloud Platform, creato il progetto, e installato Google Cloud SDK verrà eseguito il comando `gcloud auth configure-docker` che modificherà il *file config.json* di Docker in modo tale da consentire il caricamento delle immagini su Container Registry, le immagini, però, dovranno avere un nome appropriato nel formato <HOSTNAME>/<ID PROGETTO>/<NOME IMMAGINE> dove l'*hostname* indica la zona geografica dove verrà caricata l'immagine. Per quanto riguarda l'immagine SQL Server verrà caricata l'immagine su Kernel Linux.

Figura 4.12: Configurazione di Docker per Contaier Registry

```
C:\WINDOWS\system32>gcloud auth configure-docker
Adding credentials for all GCR repositories.
WARNING: A long list of credential helpers may cause delays running 'docker build'. We recommend passing the registry name to configure only the registry you are using.
After update, the following will be written to your Docker config file
 located at [C:\Users\Gerri\.docker\config.json]:
 {
  "credHelpers": {
    "gcr.io": "gcloud",
    "us.gcr.io": "gcloud",
    "eu.gcr.io": "gcloud",
    "asia.gcr.io": "gcloud",
    "staging-k8s.gcr.io": "gcloud",
    "marketplace.gcr.io": "gcloud"
  }
}

Do you want to continue (Y/n)?  Y

Docker configuration file updated.

C:\WINDOWS\system32>docker tag bike-web:windows us.gcr.io/bike-gke/bike-web:official

C:\WINDOWS\system32>docker tag bike-server:linux us.gcr.io/bike-gke/bike-server:linux
```

Tramite il comando `docker push` le immagini verranno caricate nel Container Registry del progetto indicato e rese disponibili per l'utilizzo.

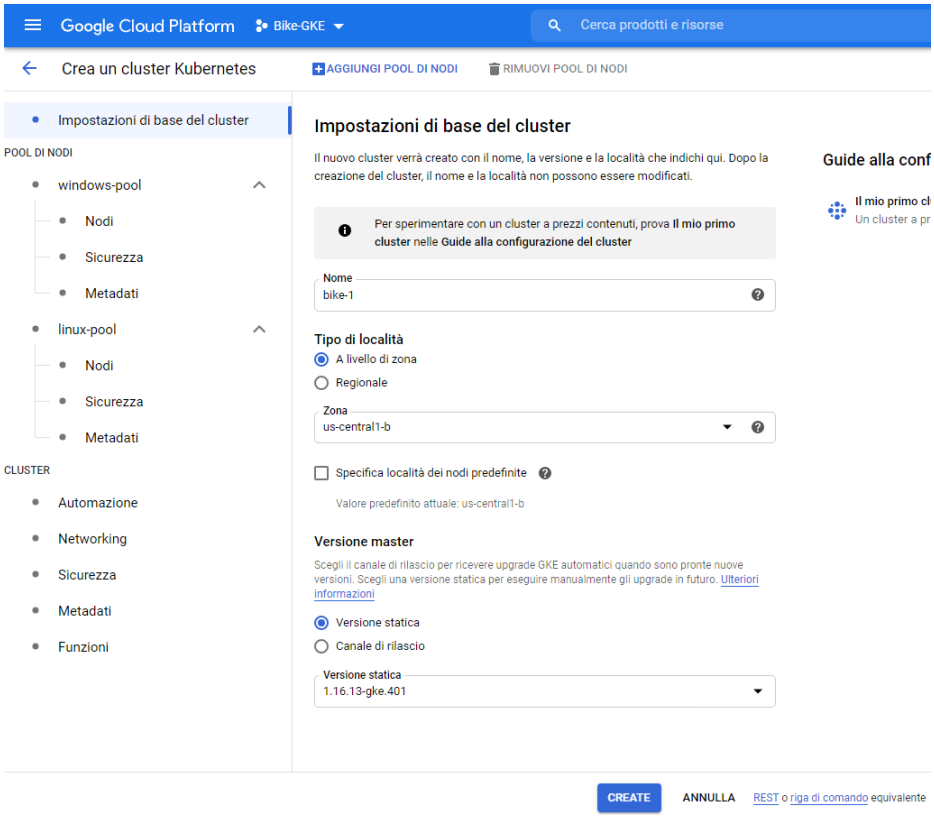
Figura 4.13: Caricamento delle immagini su Contaier Registry

```
C:\WINDOWS\system32>docker push us.gcr.io/bike-gke/bike-server:linux
The push refers to repository [us.gcr.io/bike-gke/bike-server]
7becd0fddeb9: Layer already exists
2c2dfd6f1fa6: Layer already exists
79b07b78f674: Layer already exists
be08dc7c2161: Layer already exists
911ad08015cf: Layer already exists
d33a8ef9dea5: Layer already exists
2de391e51d73: Layer already exists
d73dd9e65295: Layer already exists
686245e78935: Layer already exists
d7ff1dc646ba: Layer already exists
644879075e24: Layer already exists
linux: digest: sha256:bfd0384df3a0e8865fd597938e199832ad643883cd12d58118c97ca9b7a87836 size: 2621

C:\WINDOWS\system32>docker push us.gcr.io/bike-gke/bike-web:official
The push refers to repository [us.gcr.io/bike-gke/bike-web]
d3102fbeca39: Pushed
e9a09d063902: Pushing [====================================>              ]  33.24MB/43.22MB
545d729edba8: Pushed
1447e4ff2da9: Pushed
fc0712360783: Pushed
f9021ad367cb: Pushing [==>                                                ]  9.419MB/163.7MB
814025f87723: Pushing [>                                                  ]  1.066MB/130.1MB
662850516e74d: Pushing [================================================>]  54.27kB
2632d125814d: Waiting
25d0f1fb97e0: Waiting
c0fba5f89e4b: Waiting
54c5b86f181a: Waiting
bf734a4b22c4: Waiting
official: digest: sha256:775c764247b83ed0333250c02e0a91e903cc61a92433ca64954403da35090308 size: 3414
```

A questo punto si passa a definire il Cluster su Google Kubernetes Engine. Al momento della creazione sarà possibile scegliere:

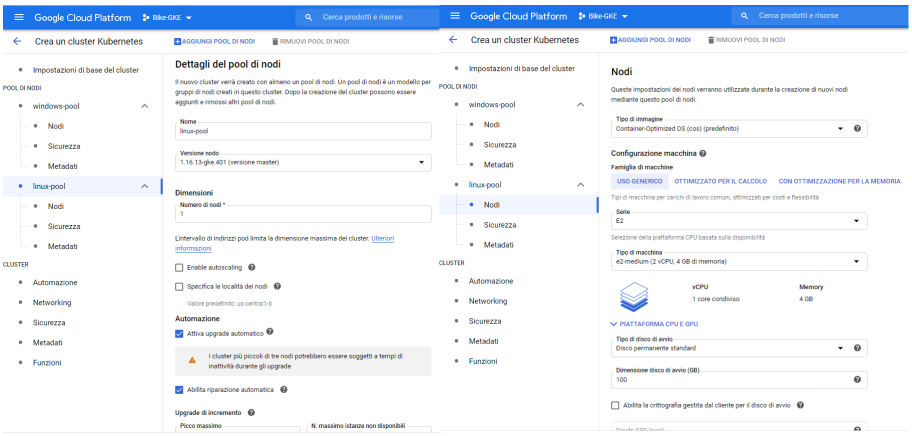
- la versione di Kubernetes da utilizzare e la zona geografica su cui si vuole lavorare;
- quanti pool di nodi utilizzare, scegliendo il numero di nodi da creare all'avvio del Cluster;
- attivare l'*upgrade automatico* per mantenere i nodi aggiornati all'ultima versione di Kubernetes e la riparazione automatica in caso di errore.

Figura 4.14: Creazione del Cluster



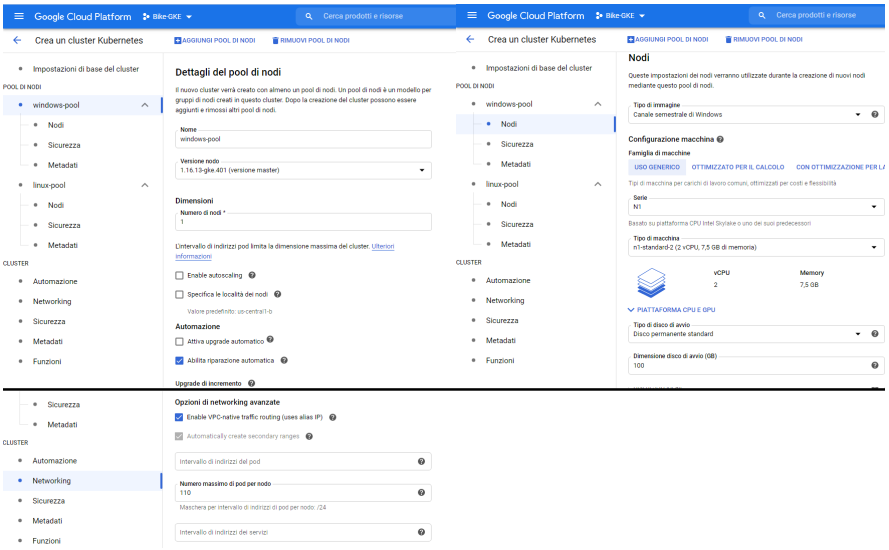
Andando più nel dettaglio, nella sezione Nodi è possibile selezionare il tipo di sistema operativo dei nodi scegliendo se essere Linux o Windows, e scegliendo il canale di rilascio delle versioni in base al tipo di lavoro da svolgere con le macchine, passando poi alle specifiche hardware come la versione della macchina, quanta RAM e CPU utilizzare, le dimensioni e il tipo del disco di avvio SSD oppure HDD.

Figura 4.15: Impostazioni del pool di Linux



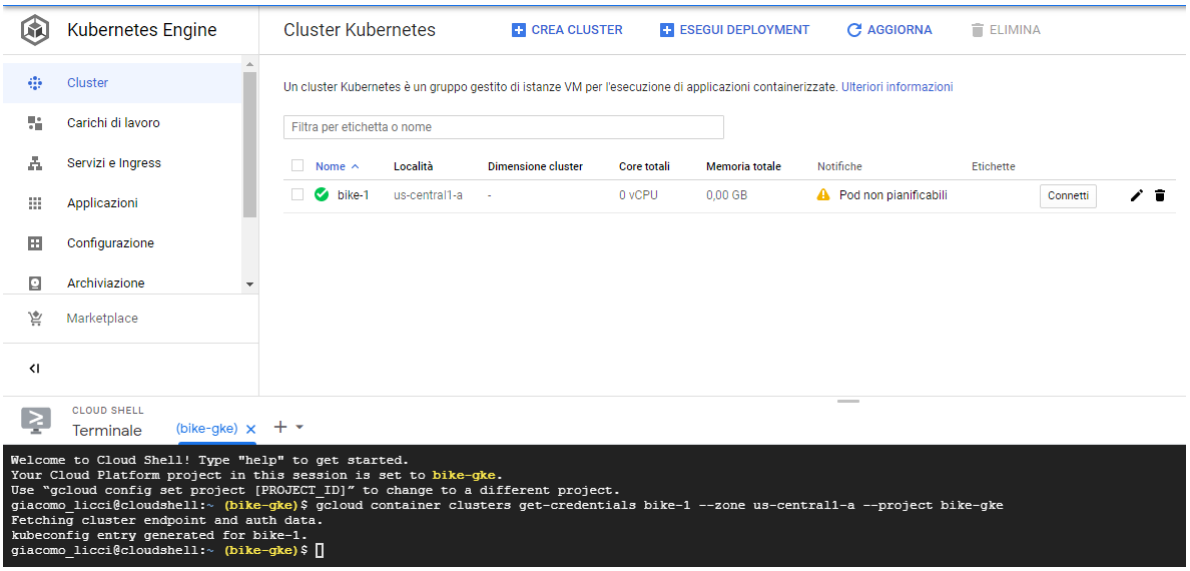
Alla creazione del Cluster è già impostato un pool di nodi Linux di default, quando si vuole creare un pool di nodi Windows basta scegliere tra "Canale semestrale di Windows" o "Long-Term Servicing Channel di Windows" nel tipo di sistema operativo, ma le macchine da utilizzare devono essere almeno del tipo "n1-standard-2" o superiori, nella sezione "Automazione" deve essere disabilitato l'*upgrade automatico* nel nodo e nella sezione "Networking" deve essere attivato il "VPC-native traffic routing".

Figura 4.16: Impostazioni del pool di Windows



Selezionate le impostazioni opportune GKE avvierà il processo di creazione tramite il tasto "Create" e una volta che le macchine all'interno dei pool saranno avviate e operative sarà possibile connettersi al Cluster con il tasto "Connetti" ed eseguire su *Cloud Shell*, i comandi per la creazione degli oggetti.

Figura 4.17: Accesso al Cluster



4.2.2 Ripristino della base di dati

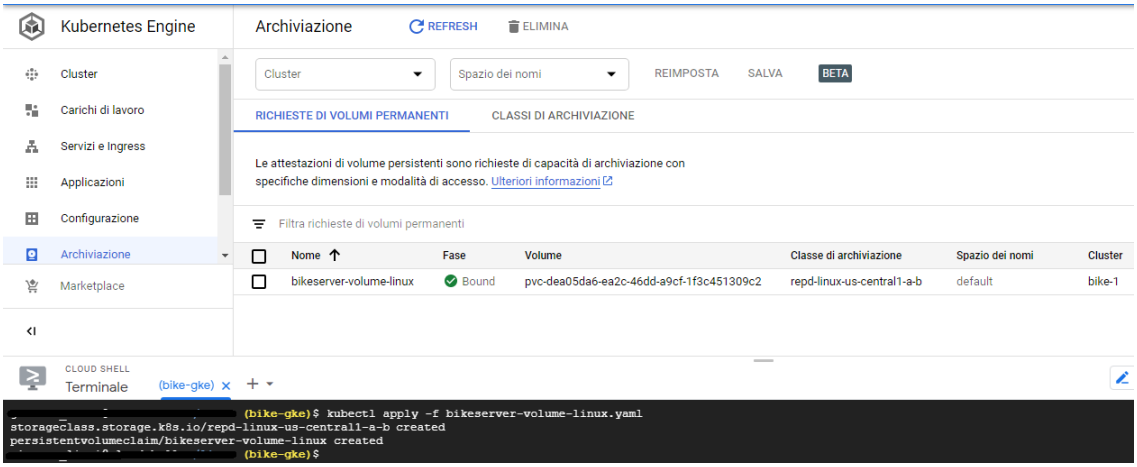
Ora che l'ambiente di sviluppo è pronto al suo utilizzo possiamo definire tutti gli oggetti per ricreare il *software* Bike tramite *Cloud Computing*. Esattamente come la creazione delle immagini partiamo col definire gli oggetti per ricreare la base di dati, primo fra questi, il volume attraverso due oggetti, StorageClass e PersistentVolumeClaim. Il primo oggetto serve per specificare il tipo di volume in base al *plugin* utilizzato a seconda del servizio su cui è eseguito Kubernetes e le zone dove verranno create le repliche del PersistentVolume, il secondo corrisponde al volume vero e proprio in cui viene indicata, oltre al tipo di StorageClass utilizzato, la sua capacità di archiviazione e l'accessMode per indicare se il volume viene montato per sola lettura da più nodi (ReadOnlyMany) o lettura e scrittura da uno o più nodi (ReadWriteOnce o ReadWriteMany) .

Figura 4.18: Definizione del volume in bikeserver-volume-linux.yaml

```
# Storage Class
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: repd-linux-us-central1-a-b
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  replication-type: regional-pd
  zones: us-central1-a, us-central1-b
---
#Volume Claim
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: bikeserver-volume-linux
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 200Gi
  storageClassName: repd-linux-us-central1-a-b
```

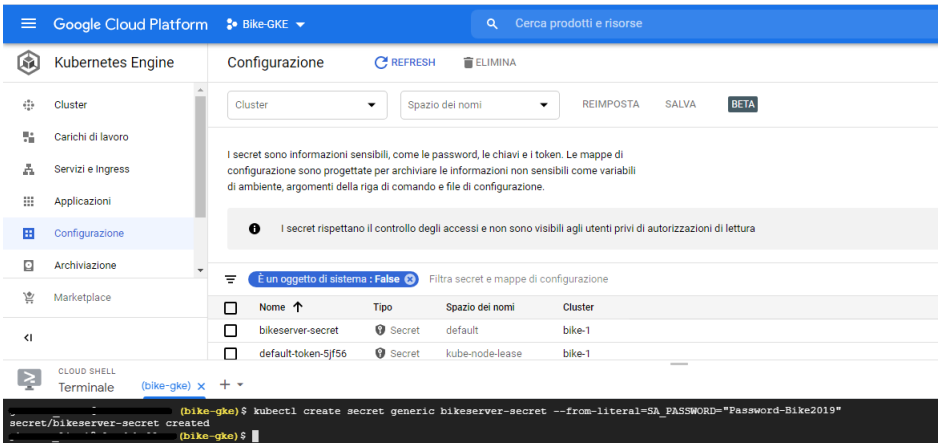
Tramite il comando `kubectl apply` andiamo a creare il volume appena definito dal *file* specificato con l'opzione `-f` verificando poi nella sezione "Archiviazione" di GKE che il volume sia pronto all'uso.

Figura 4.19: Creazione del volume in GKE



Utilizzando lo stesso comando per la creazione del volume, è possibile ricreare la base di dati, non prima di avere definito un Secret per la password del Super Amministratore di SQL Server, In questo modo possiamo definire il Deployment senza il rischio di esporre la password della base di dati e quando servirà utilizzare quel dato basterà inserire il nome del Secret e la chiave corrispondete al valore desiderato.

Figura 4.20: Creazione del Secret per base di dati



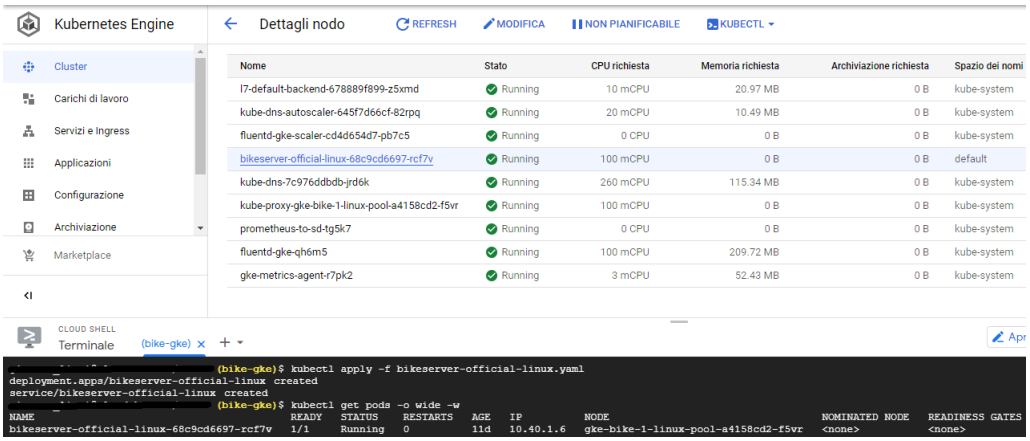
Attraverso un Deployment verrà ricreata la base di dati specificando, oltre al volume creato, le variabili ambiente e la porta che venivano passate alla creazione del container in Docker, con la possibilità i creare più repliche dello stesso container. Oltre al Deployment verrà creato un Service per rendere il Deployment, e quindi, i container della base di dati accessibili attraverso una porta specifica.

Figura 4.21: Deployment e Service in bikeserver-official-linux.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: bikeserver-official-linux
spec:
  replicas: 1
  selector:
    matchLabels:
      app: bikeserver-official-linux
  template:
    metadata:
      labels:
        app: bikeserver-official-linux
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: bikeserver-official-linux
        image: us.gcr.io/bike-gke/bike-server:linux
        ports:
        - containerPort: 1433
        env:
        - name: ACCEPT_EULA
          value: "Y"
        - name: SA_PASSWORD
          valueFrom:
            secretKeyRef:
              name: bikeserver-secret
              key: SA_PASSWORD
        volumeMounts:
        - name: bikeserver-volume-linux
          mountPath: /var/opt/mssql
      volumes:
      - name: bikeserver-volume-linux
        persistentVolumeClaim:
          claimName: bikeserver-volume-linux
```

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: bikeserver-official-linux
spec:
  selector:
    app: bikeserver-official-linux
  ports:
    - protocol: TCP
      port: 1433
      targetPort: 1433
  type: LoadBalancer
```

Figura 4.22: Creazione Deployment e Service della base di dati

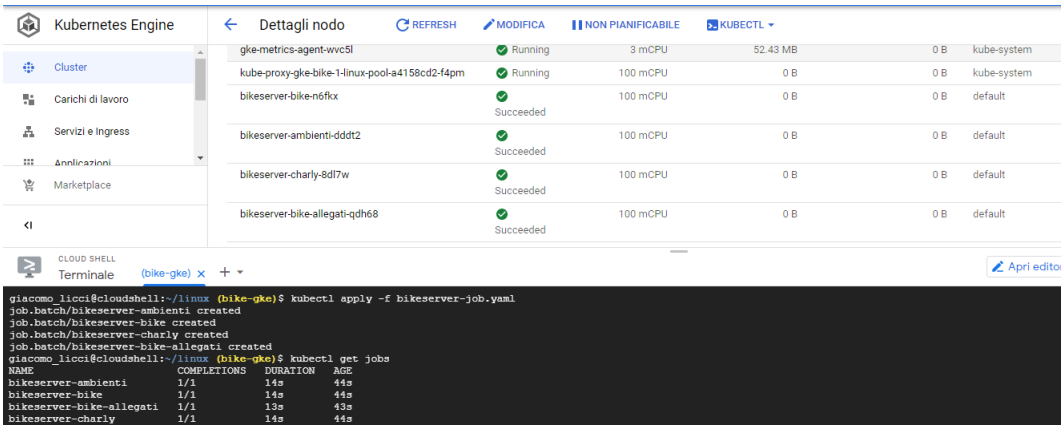


Come ultimo passaggio resta soltanto da ripristinare le tabelle della base di dati, svolto da una serie di Job, uno per ciascun *database*, che eseguiranno la `query` per il ripristino tramite il comando `sqlcmd`. Ogni Job differisce dall'altro dal nome del *database* e dal nome dei *file* utilizzati per il ripristino.

Figura 4.23: Job per ripristino Database in bikeserver-job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: bikeserver-ambienti
spec:
  template:
    spec:
      containers:
      - name: bikeserver-official-ambienti
        image: microsoft/mssql-tools
        env:
        - name: SA_PASSWORD
          valueFrom:
            secretKeyRef:
              name: bikeserver-secret
              key: SA_PASSWORD
        - name: AMBIENTI
          value: "CREATE DATABASE [AMBIENTI] ON (FILENAME = '/var/opt/META/BIKE/DATI/AMBIENTI_Data.mdf'),(FILENAME = '/var/opt/META/BIKE/DATI/AMBIENTI_Log.ldf') FOR ATTACH"
        command: ["/opt/mssql-tools/bin/sqlcmd"]
        args: ["-S", "bikeserver-official-linux", "-U", "SA", "-P", "$(SA_PASSWORD)", "-Q", "$(AMBIENTI)"]
      restartPolicy: Never
---
apiVersion: batch/v1
kind: Job
metadata:
  name: bikeserver-bike
spec:
  template:
    spec:
      containers:
      - name: bikeserver-official-bike
        image: microsoft/mssql-tools
        env:
        - name: SA_PASSWORD
          valueFrom:
            secretKeyRef:
              name: bikeserver-secret
              key: SA_PASSWORD
        - name: BIKE031
          value: "CREATE DATABASE [BIKE031] ON (FILENAME = '/var/opt/META/BIKE/DATI/BIKE_Data.mdf'),(FILENAME = '/var/opt/META/BIKE/DATI/BIKE_Log.ldf') FOR ATTACH"
        command: ["/opt/mssql-tools/bin/sqlcmd"]
        args: ["-S", "bikeserver-official-linux", "-U", "SA", "-P", "$(SA_PASSWORD)", "-Q", "$(BIKE031)"]
      restartPolicy: Never
```

Figura 4.24: Esecuzione dei Job



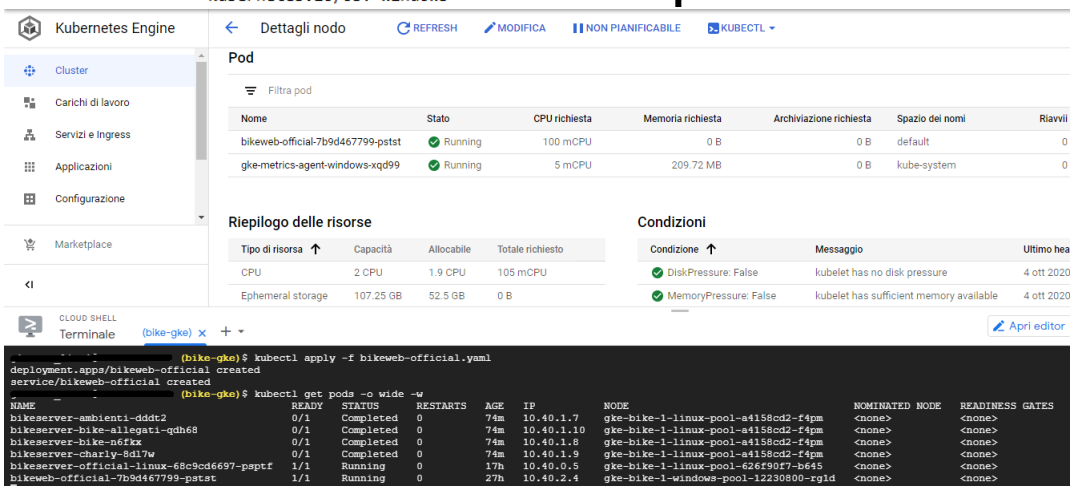
4.2.3 Ripristino dell'applicazione web

Anche per l'applicazione web di Bike verranno utilizzati un Deployment per la creazione dei container e un Service per rendere l'applicazione disponibile all'utilizzo, senza però aver bisogno del volume; per quanto riguarda l'impostazione nel *file* "ini.txt" è importante assicurarsi che in nome del server sia uguale a quello del Deployment creato per il corretto funzionamento dell'app. Un altro punto su cui soffermarsi nell'impostazione del Deployment è il valore "nodeSelector", il quale permette di scegliere i nodo dove allocare l'oggetto in base al sistema operativo. Nel Deployment della base di dati non è stato specificato perchè Kubernetes imposta di default, il valore per indirizzare gli oggetti sui nodi Linux. Per questo motivo tutti gli oggetti che lavorano su nodi Windows devono specificare il nodeSelector in modo tale da essere creati ed eseguiti sul nodo corretto.

Figura 4.25: Deployment e Service in bikeweb-official.yaml e loro creazione

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: bikeweb-official
spec:
  replicas: 1
  selector:
    matchLabels:
      app: bikeweb-official
  template:
    metadata:
      labels:
        app: bikeweb-official
    spec:
      containers:
        - name: bikeweb-official
          image: us.gcr.io/bike-gke/bike-web:official
          ports:
            - containerPort: 80
      nodeSelector:
       kubernetes.io/os: windows
```

```
---
apiVersion: v1
kind: Service
metadata:
    name: bikeweb-official
spec:
  ports:
    - name: http-port
      port: 80
      targetPort: 80
  selector:
    app: bikeweb-official
  type: LoadBalancer
```



4.3 Testing

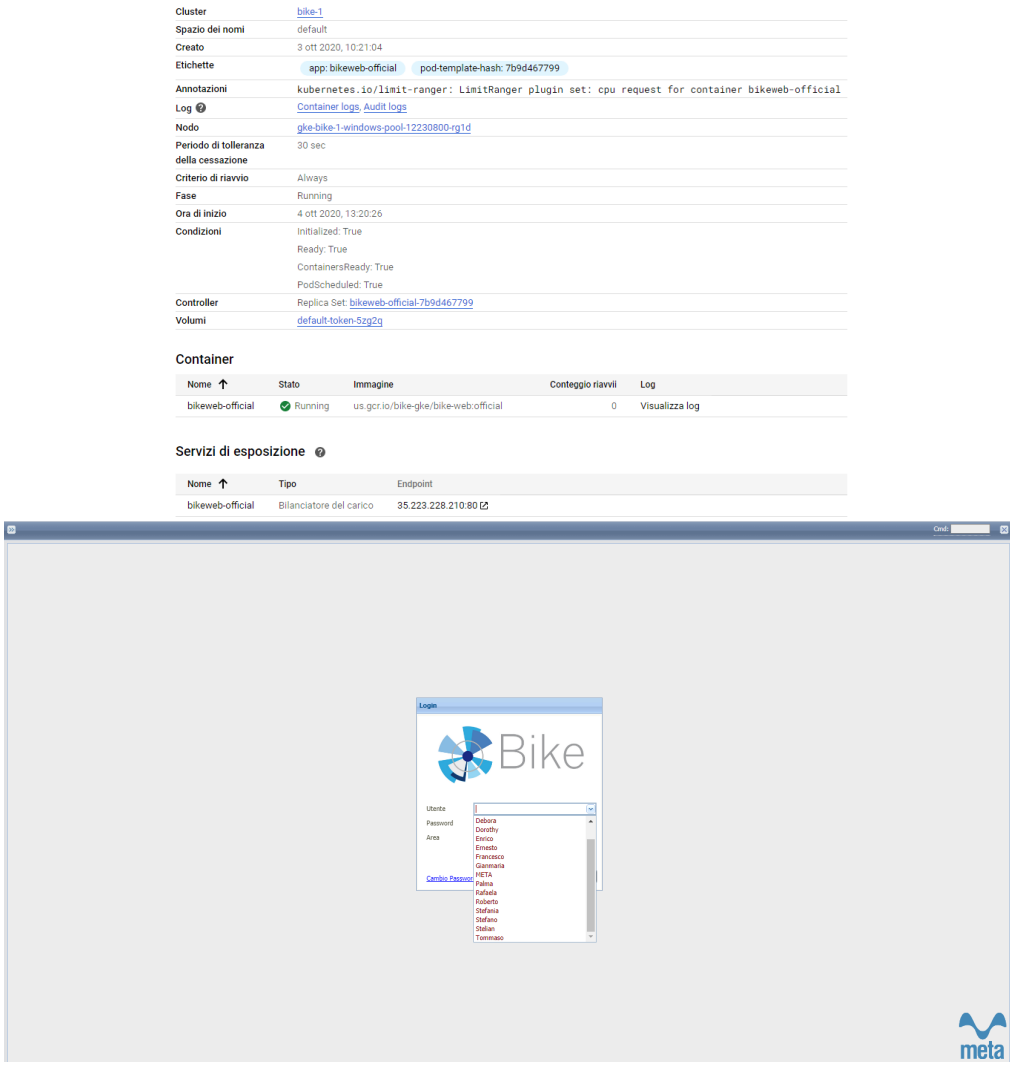
Ultimato il ripristino di ambo le parti dell'applicazione Bike si passa alla parte finale dello sviluppo che comprende il *testing* della nostra applicazione verificando che la comunicazione tra applicazione web e base di dati avvenga con successo.

4.3.1 Port Forwarding

Per effettuare l'accesso all'applicazione web basterà cliccare il nome del Pod creato tramite Deployment (Fig. 2.10) e digitare l'indirizzo IP nella sezione "Servizi di esposizione", l'accesso è consentito solo all'interno di GKE trattandosi di un indirizzo IP all'interno di una LAN privata. Se il collegamento avviene con successo, dopo un breve caricamento apparirà la schermata di *login* con la possibilità di scegliere l'utente attraverso un menu a tendina e digitare la password per l'accesso. Nel nostro caso utilizzeremo l'utente META che corrisponde all'utente amministratore, in modo tale da accedere a tutte le funzioni disponibili all'uso,

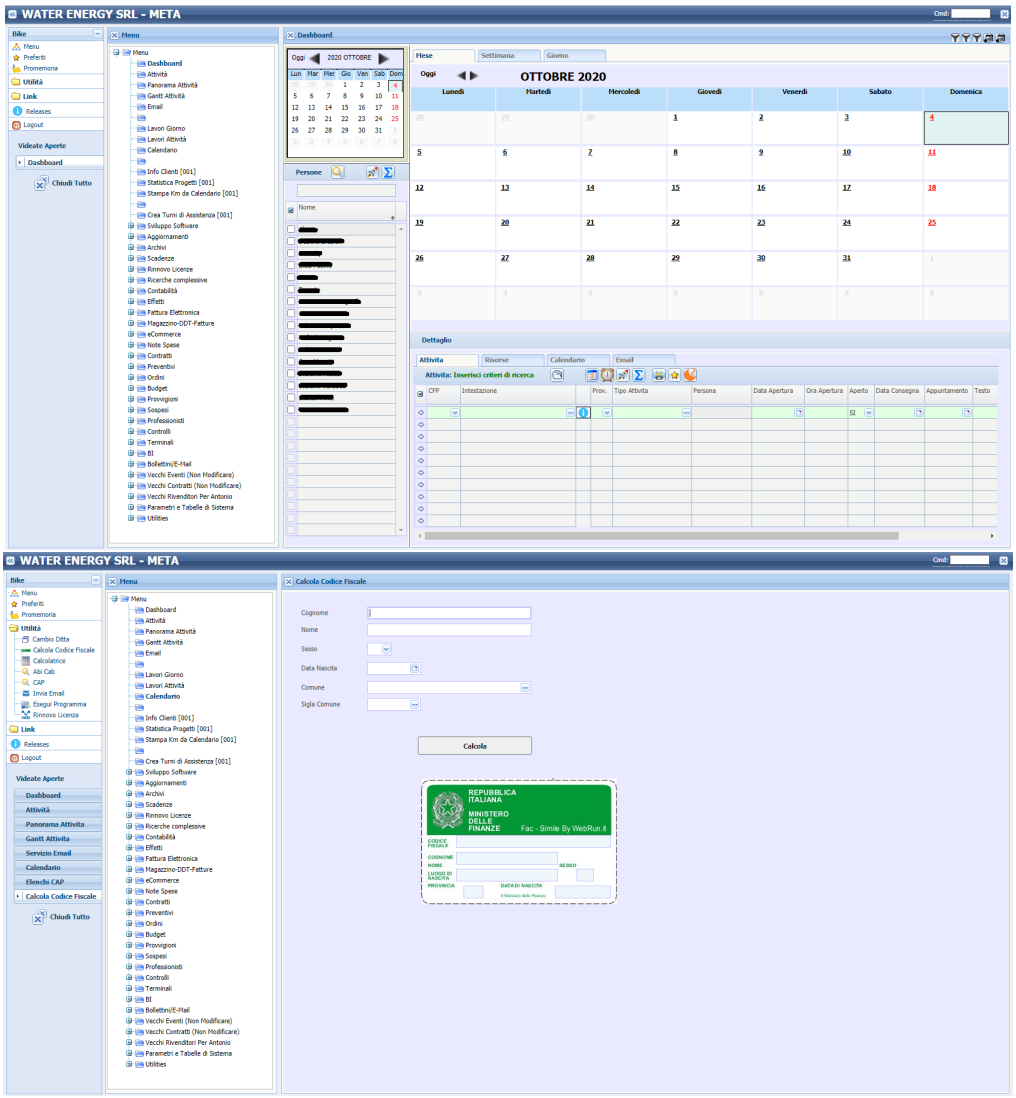
eccezion fatta per quelle disabilitate o ancora in via di sviluppo.

Figura 4.26: Accesso all'applicazione e schermata login



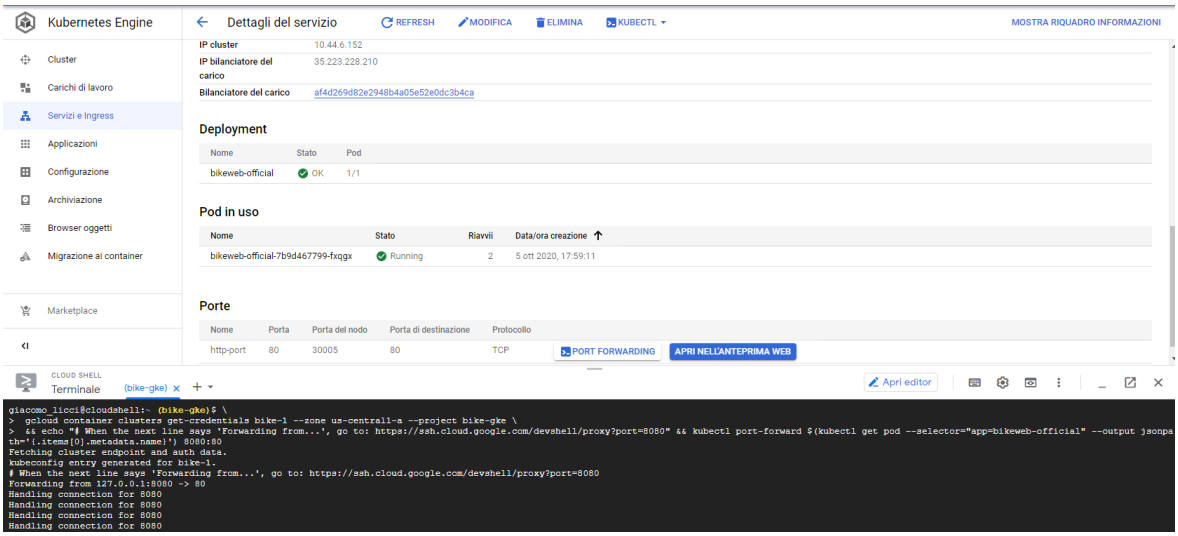
Una volta inseriti i dati di accesso si verrà indirizzati alla pagina principale di Bike dove potremmo accedere ai servizi di Bike attraverso un menù, con la possibilità di aprire e spostarsi su più sotto-finestre contemporaneamente. Cliccando nella cartella "Utilità" è possibile accedere a tutte le funzioni di supporto per l'utente come il calcolo automatico del Codice Fiscale, la ricerca del CAP, l'invio delle e-mail e il rinnovo della licenza Bike.

Figura 4.27: Funzioni di Bike



Accertato il corretto funzionamento dell'applicazione si procede con il *Port Forwarding* dell'applicazione, in modo tale che gli utenti esterni possano accedere a Bike attraverso la porta specificata nel Service senza entrare su GKE. Per attivare il *Port Forwarding* basta entrare nei dettagli del servizio dell'applicazione web e cliccare sul tasto "Port Forwarding" per eseguire il comando su *Cloud Shell*.

Figura 4.28: Port Forwarding



4.3.2 Problematiche incontrate

La principale problematica incontrata nel progetto di tesi è la compatibilità delle immagini create in Docker e le macchine Windows di GKE. Anche se si utilizza l'ultima versione di Kubernetes e viene creato un Pod con un'immagine Windows Server superiore alla versione 1909, il container andrà in errore. È necessario specificare nel Dockerfile una versione di Windows Server uguale o precedente alla 1909. Sempre in merito alla compatibilità Windows un altro problema è insorto con l'immagine di SQL Server e la versione di Windows Server, nonostante l'avvio del container veniva svolto correttamente, non era possibile eseguire il collegamento al *server*, rendendo impossibile eseguire il comando di ripristino della base di dati e l'interrogazione da parte dell'applicazione web. Si è quindi optato per utilizzare un'immagine su Kernel Linux avendo funzionamento analogo a quella su Kernel Windows.

Capitolo 5

Conclusioni

L'elaborato ha illustrato lo sviluppo del progetto di tesi partendo dall'introduzione e lo studio dei metodi e strumenti necessari, fino alla loro implementazione.

L'obbiettivo del progetto consisteva nel riprogettare un'applicazione web utilizzando macchine virtuali a basso utilizzo di risorse, attraverso un servizio di *Cloud Computing*. Gli strumenti e le metodologie applicate hanno permesso di rispettare tutti i requisiti progettuali, uno tra questi la scelta di utilizzare Docker per la creazione delle immagini. Infatti, Docker permette di usufruire di un gran numero di immagini di ogni genere e tipo, sia che lavorino su Kernel Linux che su Kernel Windows, dando la possibilità di personalizzare le immagini a seconda del servizio da "containerizzare".
I servizi Google Kubernetes Engine e Container Registry, offerti dalla piattaforma Google Cloud e utilizzati in coppia, hanno permesso di creare facilmente l'ambiente di sviluppo per Bike, archiviando le immagini create con Docker e permettendo di creare facilmente il Cluster con i rispettivi nodi, per l'utilizzo degli oggetti di Kubernetes, permettendo di controllare il funzionamento attraverso un'interfaccia semplice e facile da usare. Anche se GKE ha riscontrato problemi di compatibilità con le macchine Windows, è stato comunque possibile sviluppare ed eseguire l'applicazione correttamente scegliendo la versione corretta di Windows Server per le immagini Windows o, in alternativa, utilizzando lo stesso tipo di immagini, ma implementate su Kernel Linux.

In conclusione nonostante si siano riscontrati alcuni problemi di compatibilità, lo sviluppo dell'applicazione Bike ha soddisfatto i requisiti iniziali.

5.1 Sviluppi Futuri

Nonostante gli strumenti e i metodi utilizzati abbiano consentito la ri-progettazione di un sistema ERP tramite container permettendo il corretto funzionamento e l'accesso al servizio a tutti gli utenti, sono stati utilizzati due nodi con SO(Sistema Operativo) diversi, piuttosto che due con SO identico.
Di conseguenza, quando é necessario creare o riavviare i nodi su cui risiede l'applicazione, bisogna attendere che i nodi con lo stesso SO finiscano il processo di creazione per poi procedere con la creazione dei nodi dell'altro tipo. Se, invece, si utilizzano nodi con stesso SO, in questo caso Windows, basta aspettare i completamento del processi di creazione e l'applicazione sarà pronta al suo utilizzo.
Una possibile implementazione futura è riuscire a correggere i problemi riscontrati con l'immagine Windows della base di dati in modo tale da utilizzare solo nodi Windows.
Un'altra implementazione possibile è quella di creare più repliche di basi di dati e di applicazione web per ciascuna azienda che utilizza Bike, in modo tale da avere una visione e un controllo centralizzato sullo stato di tutte le macchine consentendo un assistenza rapida e semplice sul servizio offerto.

Capitolo 6

Ringraziamenti

A conclusione di questo elaborato é doveroso menzionare tutte le persone che hanno permesso la realizzazione di questo progetto di tesi. Ringrazio, in primis, il mio relatore Mancini Adriano per avermi proposto questo interessante progetto e che ha saputo guidarmi e sostenermi in questi mesi di lavoro, attraverso suggerimenti per apprendere nel migliore dei modi il funzionamento degli strumenti utilizzati e per risolvere tutte le problematiche incontrate durante lo sviluppo.
Ringrazio inoltre l'azienda Meta Informatica S.R.L. e il titolare Lucarelli Paolo per avermi consentito di utilizzare il loro software per lo sviluppo del progetto, ringrazio inoltre tutti i dipendenti, in particolar modo, il mio supervisore Bergami Tommaso e tutto il reparto hardware e sviluppo software, per avermi aiutato a comprendere il funzionamento dell'applicazione Bike e fornendomi tutto il materiale necessario per la sua ri-progettazione. Ultimo ma non meno importante, vorrei ringraziare tutta la mia famiglia per avermi sempre appoggiato e spronato nel percorso di studi che ho scelto, dandomi manforte nei momenti di difficoltà, più di ogni altra cosa.

Bibliografia

[1] https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet
ultima visita 29/09/2020.

[2] https://www.erpselection.it/software/bike/
ultima visita 29/09/2020.

[3] Kelsey Higtower Joe Beda Brendan Burns. *Kubernetes Up and Running: Dive into the future of the infrastructure*. O'Reilly, 2017.

[4] https://www.redhat.com/it/topics/devops/what-is-ci-cd
ultima visita 25/09/2020.

[5] https://www.redhat.com/it/topics/containers/what-is-a-kubernetes-cluster/
ultima visita 25/09/2020.

[6] https://www.docker.com/resources/what-container
ultima visita 23/09/2020.

[7] https://it.wikipedia.org/wiki/docker
ultima visita 23/09/2020.

[8] https://docs.docker.com/engine/reference/builder/
ultima visita 25/09/2020.

[9] https://www.redhat.com/it/topics/containers/what-is-docker
ultima visita 23/09/2020.

[10] https://www.centrosoftware.com/cose-un-sistema-erp-di-ultima-generazione
ultima visita 22/09/2020.

[11] https://en.wikipedia.org/wiki/google_cloud_platform
ultima visita 26/09/2020.

[12] https://cloud.google.com/kubernetes-engine
ultima visita 27/09/2020.

[13] https://cloud.google.com/kubernetes-engine/docs/concepts/kubernetes-engine-overview
ultima visita 26/09/2020.

[14] https://www.instantdeveloper.com/
ultima visita 01/10/2020.

[15] https://www.instantdeveloper.com/vantaggi
ultima visita 01/10/2020.

[16] https://kubernetes.io/it/docs/concepts/overview/what-is-kubernetes/
ultima visita 25/09/2020.

[17] Meta informatica s.r.l. https://metainformatica.com/, 2020.

[18] https://doc.instantdeveloper.com/267004d3-71db-0ebe-1b2e-e498929fe1bc.htm?lang=ita
ultima visita 01/10/2020.

[19] https://www.microsoft.com/it-it/sql-server/sql-server-2019
ultima visita 14/10/2020.

[20] https://www.smeup.com/blog/blog-software-gestionali-erp/sistema-erp-definizione-vantaggi/: :text=riduzione%20del%20rischio%20di%20 duplicazione,e%20miglior%20reperimento%20dei%20dati
ultima visita 27/09/2020.

[21] https://kubernetes.io/docs/concepts/storage/volumes
ultima visita 26/09/2020.

[22] https://www.html.it/pag/62783/docker-e-i-container/
ultima visita 15/10/2020.

[23] https://www.oracle.com/it/applications/erp/what-is-erp.html
ultima visita 27/09/2020.

Elenco delle figure