



Relazione progetto di Sistemi Cloud

Creazione di una chat multiutente orchestrata
da kubernetes su VM Azure

Stefano Colombo
Matricola 1000024706

Indice

1	Introduzione	2
2	MariaDB	5
3	RabbitMQ	6
4	Spring Boot	7
4.1	Frontend	7
4.2	Backend	10
4.3	Frontend e Backend file yaml	12
5	Creazione Macchine Virtuali Azure	14
6	Configurare le VM con Kubernetes e far partire i deployment	16

1 Introduzione

L'elaborato prodotto mostra lo sviluppo di una chat asincrona multiutente orchestrata da kubernetes su macchine virtuali della piattaforma cloud Azure. Viene mostrato come la chat viene implementata tramite un'architettura a microservizi, eseguiti in container Docker. Il progetto è composto da 4 container: un container Rabbitmq, per la creazione e la gestione delle code, utilizzate per l'invio di messaggi; un container Mariadb, per la memorizzazione dello storico della chat e due container OpenJDK che eseguono applicazioni Spring Boot che contengono la logica del funzionamento della chat.

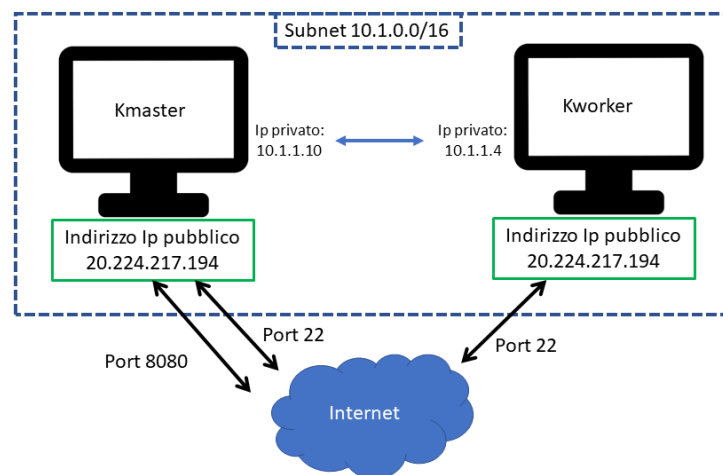


Figura 1: Configurazione della rete

La configurazione di rete risulta essere molto semplice, si utilizzano due macchine virtuali Azure, una avrà il compito di Master e l'altra ospiterà i pod eseguendo il ruolo di worker (volendo si possono aggiungere tranquillamente anche altre VM worker); le VM sono situate all'interno della stessa subnet, così da poterle fare dialogare tranquillamente. Entrambe le macchine hanno un indirizzo IP pubblico necessario per permettere il collegamento da remoto, entrambe espongono la porta 22 per le connessioni ssh e il kmaster in più espone anche la porta 8080 per fornire l'interfaccia web agli utenti.

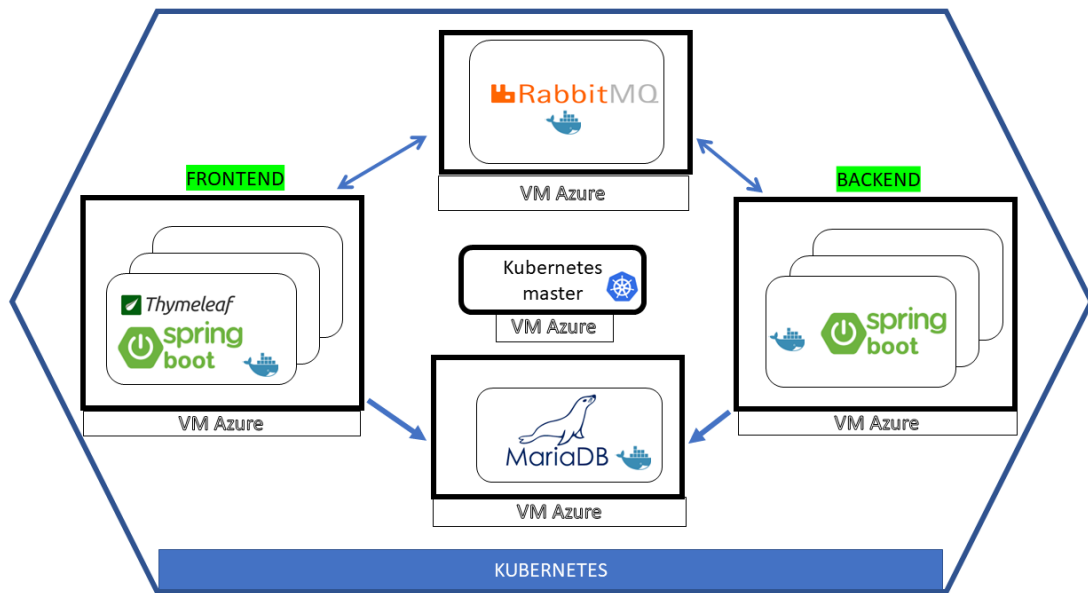


Figura 2: Collegamento servizi

I collegamenti tra i servizi sono mostrati nell'immagine sopra. il Frontend carica lo storico dei messaggi richiedendolo al servizio MariaDB e invia/legge messaggi dal servizio RabbitMQ mentre il Backend memorizza i nuovi messaggi nel Servizio MariaDB e legge/invia messaggi al servizio RabbitMQ.

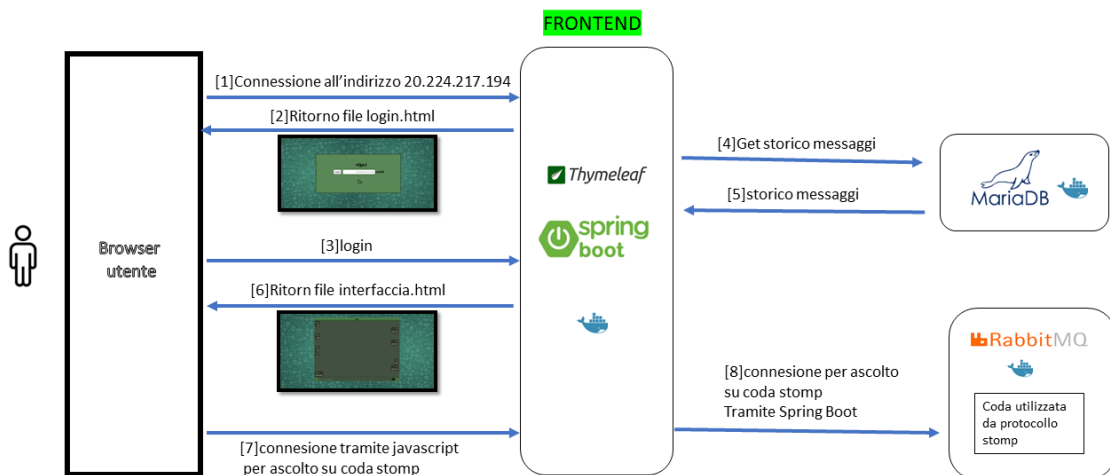


Figura 3: login utente

L'utente si collega all'indirizzo 20.224.217.194 sulla porta 8080 (1); il frontend risponde inviando la pagina login.html, grazie al RestController, implementato tramite Spring Boot, che ritorna pagine html create tramite Thymeleaf(2); l'utente esegue il login(3); il frontend chiede lo storico dei messaggi al servizio MARIADB grazie al framework JPA configurato su spring Boot(4); Il DB risponde inviando lo storico dei messaggi (5); Spring

Boot ritorna la pagina interfaccia.html che conterrà tutti i messaggi inviati precedentemente e il codice javascript della pagina si metterà in ascolto, tramite il (7) frontend, grazie al protocollo STOMP, del servizio RabbitMQ.

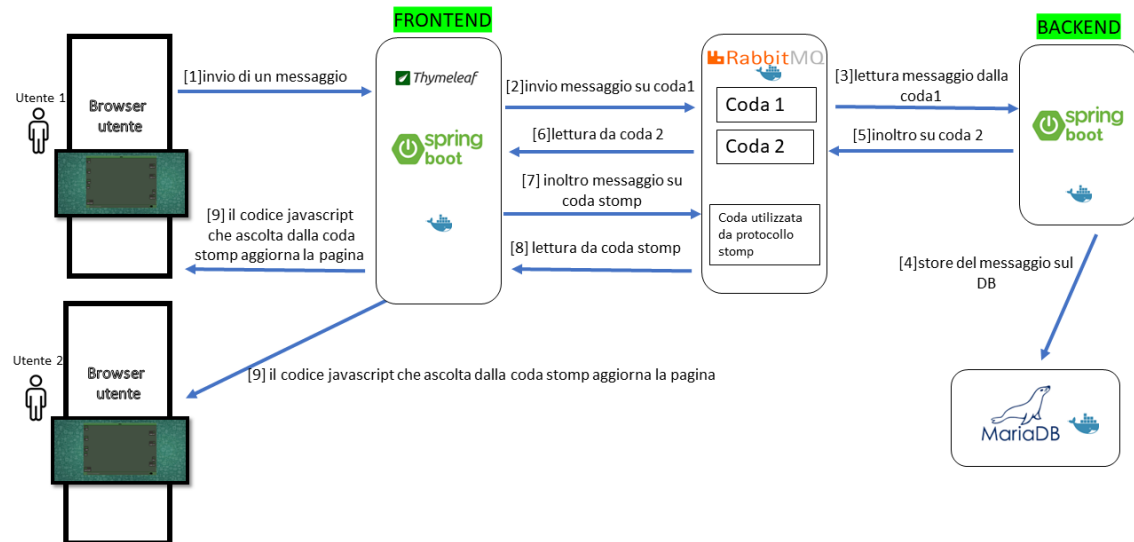


Figura 4: Invio messaggi

L'utente1 invia un messaggio(1); il frontend invia il messaggio sulla coda1 di RabbitMQ tramite il protocollo AMQP di Spring Boot(2); il backend legge i messaggi dalla coda1(3); memorizza i messaggi nel database MariaDB grazie a JPA(4); inoltra i messaggi sulla coda2(5); il frontend legge i messaggi dalla coda2(6); il frontend li inoltra a sua volta sulla coda utilizzata dal protocollo STOMP(7); e il codice javascript nell'interfaccia utente legge i messaggi della coda STOMP(9), tramite Spring Boot del frontend(8), e aggiorna la pagina html di tutti gli utenti connessi in quel momento.

Nei capitoli successivi verrà mostrato inizialmente come sono stati implementati e configurati i servizi RabbitMQ, MariaDB, Frontend e Backend; si spiegherà come sono state create e configurate le VM con all'interno kubernetes e come far partire l'applicazione. Le configurazioni dei servizi verranno mostrate con la sintassi dei file .yaml che riesce a leggere il tool docker-compose, questo perché una volta che viene creato il file .yaml si utilizzerà il tool Kompose che permette di convertire i file in una sintassi che kubernetes riesce a leggere.

2 MariaDB

Il container MariaDB viene utilizzato per memorizzare lo storico delle chat all'interno di un database. Il database presenta una sola tabella dove ogni record rappresenta un messaggio. I record vengono registrati una volta che il messaggio arriva a destinazione e non prima, così facendo si evita di popolare il database di messaggi che vengono inviati ma che, magari per errori dovuti alla rete, non riescono a raggiungere la destinazione.

Field	Type	Null	Key	Default	Extra
id	bigint(20)	NO	PRI	NULL	auto_increment
contenuto	varchar(255)	YES		NULL	
mittente	varchar(255)	YES		NULL	
tipo	varchar(255)	YES		NULL	

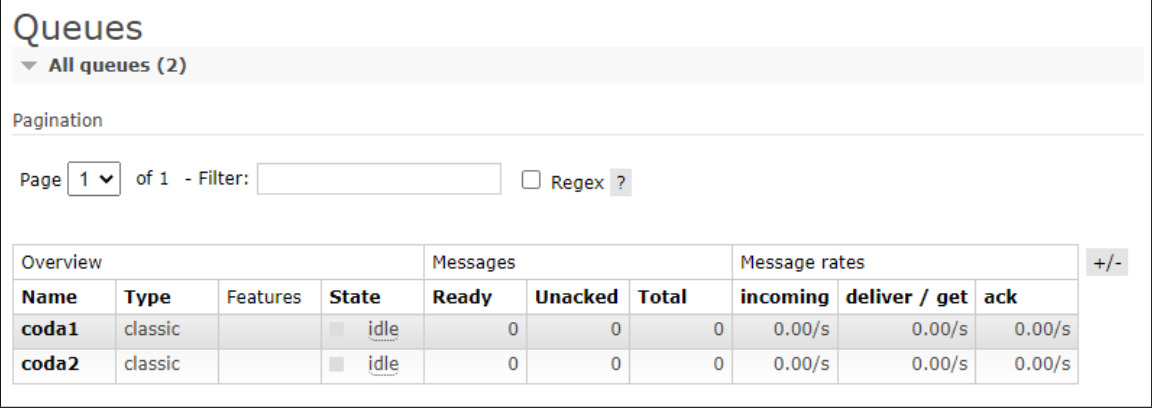
Figura 5: Tabella database

Nel file .yml il container Mariadb presenta le seguenti configurazioni: con l'attributo "image" si indica l'immagine di base del container, in questo caso risulta essere "mariadb:latest"; con "ports" il container mappa la porta 3306 con la porta 3006; con "volumes" viene mappato il contenuto della cartella "/var/lib/mysql/" nel volume di nome "db-data", questa configurazione permettere al contenuto del database di sopravvivere dopo la distruzione del container; e tramite "enviroment" si definiscono le variabili d'ambiente che indicano rispettivamente: la password di root, il nome del database da utilizzare, il nome dell'utente e la password dell'utente.

```
...
mariadb:
  image: mariadb:latest
  ports:
    - "4002:3306"
  volumes:
    - db-data:/var/lib/mysql/
  enviroment:
    - MYSQL_ROOT_PASSWORD=1234
    - MYSQL_DATABASE=chat
    - MYSQL_USER=stefano
    - MYSQL_PASSWORD=ste
...
```

3 RabbitMQ

Il container RabbitMQ è un broker di messaggi che viene utilizzato per la creazione e l'utilizzo di code che permettono ai due container OpenJDK di comunicare.



The screenshot shows the RabbitMQ 'Queues' management page. It displays a table with two queues, 'coda1' and 'coda2', both of type 'classic' and in an 'idle' state. The table includes columns for 'Overview' (Name, Type, Features, State), 'Messages' (Ready, Unacked, Total), and 'Message rates' (incoming, deliver / get, ack). The 'coda1' and 'coda2' queues both show 0 ready messages, 0 unacked messages, and 0 total messages, with message rates of 0.00/s.

Overview				Messages			Message rates			+/-
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
coda1	classic		idle	0	0	0	0.00/s	0.00/s	0.00/s	
coda2	classic		idle	0	0	0	0.00/s	0.00/s	0.00/s	

Figura 6: Code create

Nel file .yaml è presente un'immagine personalizzata "stecol/rabbitmq", questa immagine è stata creata attraverso il Dockerfile sottostante, che prende l'immagine "rabbitmq:management-alpine", esegue il comando indicando da "RUN" ed espone le porte mostrare; questa modifica dell'immagine risulta essere necessaria per abilitare il protocollo rabbitmq_stomp utilizzato dall'interfaccia web del servizio client. L'immagine desiderata si crea col comando "docker build . -t stecol/rabbitmq".

Nel file .yml il servizio RabbitMQ presenta le seguenti configurazioni: si basa su l'immagine "stecol/rabbitmq" ed espone le porte 5672, 15672 e 61613 . La porta 5672 serve ai container per collegarsi al servizio, la porta 15672 non serve all'utente finale che utilizza la chat ma risulta essere utile per lo sviluppo di essa, visto che permette di accedere all'interfaccia web "RabbitMQ management" che permette di controllare le code create, le connessioni al servizio e altri dati che possono essere utili; infine, la porta 61613 serve al codice javascript dell'interfaccia web per collegarsi al servizio. Le immagini sopra mostrano le code create.

```
FROM rabbitmq:management-alpine
RUN rabbitmq-plugins enable rabbitmq_stomp
EXPOSE 5672 15672 61613
```

```
...
services:
  rabbitmq:
    image: 'stecol/rabbitmq'
    ports:
      - '5672:5672'
      - '15672:15672'
      - '61613:61613'
...
```

4 Spring Boot

Spring Boot è un tool molto utile che permette di configurare e definire applicazioni Spring riducendo la complessità nel dover includere librerie, dipendenze e configurazioni che rendono arduo lo sviluppo e la messa in produzioni di applicazioni. In questo capitolo per evitare di dilungarsi troppo sulla logica della chat, verranno mostrate solo le strutture delle applicazioni e le parti di codice dove avviene la connessione con gli altri servizi.

4.1 Frontend

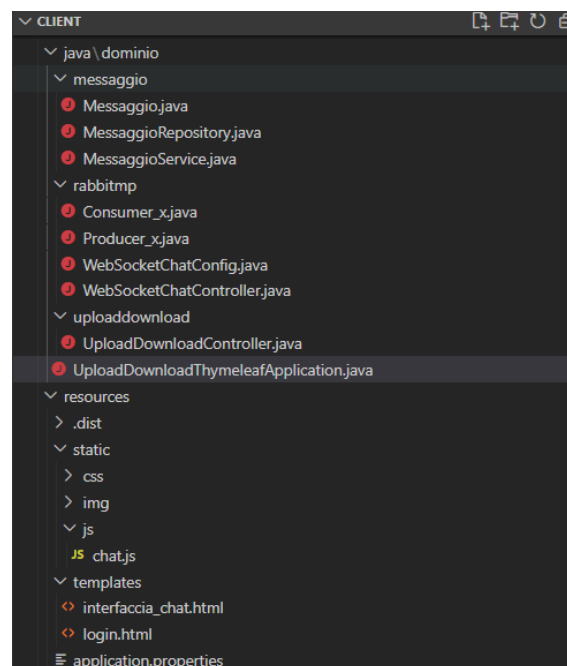


Figura 7: Struttura frontend

Le classi che compongono l'applicazione sono suddivise in package in base al loro ruolo. Le classi col prefisso "Messaggio" implementano le classi Entity, Repository e Service che permettono di dialogare con il database remoto MariaDB attraverso JPA, grazie ad esse gli oggetti di tipo Messaggio verranno mappati come record della tabella "Messaggio" sul database e ad esempio operazioni di creazioni o cancellazione degli oggetti verranno tradotte come opportune query sul database, in questo caso nel frontend ci permettono di caricare tutto lo storico della chat dopo che un utente esegue il login. Le classi del package "rabbitmq" permettono di inviare e ricevere dati dalle code RabbitMQ. La Classe UploadDownloadController svolge il ruolo di RestController che permette di mappare le chiamate Rest ad opportuni metodi, in particolar modo si occupa di richiamare i metodi per memorizzare/inviare messaggi tramite il MessaggioService e le opportune classi "rabbitmq". Il file application.properties e la classe UploadDownloadThymeleafApplication rispettivamente definiscono le configurazioni dell'applicazione e richiamano i metodi necessari per connettersi ai container MariaDB e RabbitMQ. I file "interfaccia_chat.html" e "login.html" sono le interfacce web create tramite il tool Thymeleaf che permette di

manipolare i dati ricevuti dal RestController per creare una pagina HTML. Infine il file "chat.js" permette di manipolare la pagina HTML e avvia la connessione con il servizio rabbitmq grazie alle configurazioni dei file "WebSocket" nella cartella rabbitmq.

```
#spring.datasource.url=jdbc:mysql://mariadb/chat
spring.datasource.username=stefano
spring.datasource.password=ste
spring.jpa.hibernate.ddl-auto = update

#rabbit.indirizzo=rabbitmq
coda.prodotto=coda1
coda.consumatore=coda2

#server.port=8080
```

Nel file application.properties vengono aggiunte le configurazioni per i nomi delle code e la connessione al database, i valori commentanti non vengono aggiunti in questo punto ma vengono passati come variabili d'ambiente. In questo caso si sta settando il nome e la password per accedere al database e il nome che le code avranno una volta avvenuta la connessione col broker dei messaggi rabbitmq.

```
1 ...
2 //configurazioni per la creazione della coda parte consumatore
3 @Bean
4 Queue queue(@Value("${coda.prodotto}") String queueName) {
5     return new Queue(queueName, false);
6 }
7
8 //collegamento al broker di messaggi rabbitmq
9 @Bean
10 CachingConnectionFactory connectionFactory(@Value("${rabbit.indirizzo}") String indirizzo) {
11     CachingConnectionFactory cachingConnectionFactory =
12     new CachingConnectionFactory(indirizzo);
13     return cachingConnectionFactory;
14 }
15 ...
```

Nella classe UploadDownloadThymeleafApplication avviene la connessione con il servizio rabbitmq e viene creata la coda sulla quale il client scriverà, ovvero sulla coda1.

```

1 ...
2 RabbitListener(queues = "${coda.consumatore}" )
3 public void run(Message message) throws InterruptedException,IOException {
4
5     String body=new String(message.getBody());
6     System.out.println("_\\t\\t\\t\\t\\t\\t\\t[received]:" +body+ "");
7     messagingTemplate.convertAndSend("/topic/messaggi_chat", body.toString());
8 }
9 ...

```

```

1 ...
2 private String indirizzo;
3
4 @Autowired
5 public WebSocketChatConfig (@Value("${rabbit.indirizzo}") String rabbit) {
6     this.indirizzo=rabbit;
7 }
8
9 @Override
10 public void registerStompEndpoints(StompEndpointRegistry registry) {
11     registry.addEndpoint("/websocketApp").withSockJS();
12 }
13
14 @Override
15 public void configureMessageBroker (MessageBrokerRegistry registry) {
16     registry.setApplicationDestinationPrefixes("/app");
17     registry.enableStompBrokerRelay("/topic")
18         .setRelayHost(indirizzo)
19         .setRelayPort(61613)
20         .setClientLogin("guest")
21         .setClientPasscode("guest");
22 }
23 ...

```

Nella classe WebSocketChatConfig si inseriscono le configurazioni per utilizzare il plugin_stomp che è stato aggiunto nel servizio rabbitmq; quindi, si aggiunge un endpoint e si settano tutte le configurazioni necessarie; in questo caso l'applicazione si collegherà al servizio rabbitmq tramite la porta "61613" come utente "guest" e passcode "guest". Queste configurazioni sono necessarie per comunicare con l'interfaccia web dell'utente e avvisarla quando è arrivato un nuovo messaggio.

```

1 ...
2 function connect() {
3     var socket = new SockJS('/websocketApp');
4     stompClient = Stomp.over(socket);
5     stompClient.connect({}, connectionSuccess);
6 }
7
8 function connectionSuccess() {
9     stompClient.subscribe('/topic/messaggi_chat', onMessageReceived);
10    console.log("messaggio_ricevuto1");
11 }
12 ...

```

Infine, nel file chat.js avviene la connessione con l'endpoint, definito prima, e si indica su quale topic ascoltare l'arrivo dei nuovi messaggi. La funzione onMessageReceived non viene mostrata ma ha il solo scopo di modificare HTML dell'interfaccia web per aggiungere i messaggi che arrivano.

4.2 Backend

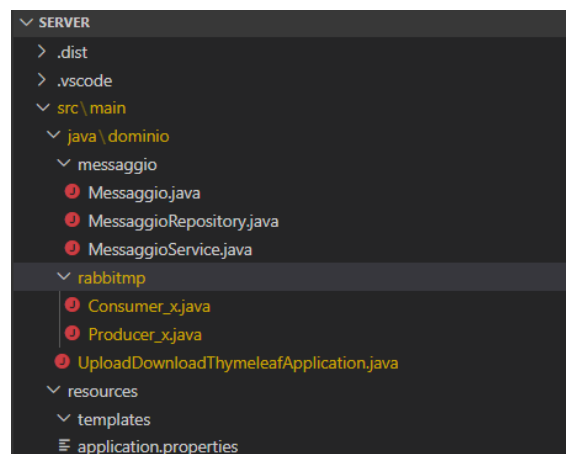


Figura 8: Struttura Backend

Il Backend si occupa solamente di rinviare i messaggi, quindi presenta una struttura più semplice rispetto al frontend. Le classi col prefisso "Messaggio" implementano le classi Entity, Repository e Service che permettono di dialogare con il database remoto MariaDB attraverso JPA, nel Backend vengono utilizzate per memorizzare i nuovi messaggi nel database. Le classi del package "rabbitmq" permettono di inviare e ricevere dati dalle code RabbitMQ. Il file application.properties e la classe UploadDownloadThymeleafApplication rispettivamente definiscono le configurazioni dell'applicazione e richiamano i metodi necessari per connettersi ai container MariaDB e RabbitMQ.

```

#spring.datasource.url=jdbc:mysql://mariadb/chat
spring.datasource.username=stefano
spring.datasource.password=ste
spring.jpa.hibernate.ddl-auto = update

#rabbit.indirizzo=rabbitmq
coda.prodotto=coda2
coda.consumatore=coda1
#server.port=8081

```

Nel file application.properties vengono aggiunte le configurati per i nomi delle code e la connessione al database, i valori commentanti non vengono aggiunti in questo punto ma vengono passati come variabili d'ambiente. A differenza del frontend qua i nomi delle code sono invertiti.

```

1 ...
2 //configurazioni per la creazione della coda parte consumatore
3 @Bean
4 Queue queue(@Value("${coda.prodotto}") String queueName) {
5     return new Queue(queueName, false);
6 }
7 //collegamento al broker di messaggi rabbitmq
8 @Bean
9 CachingConnectionFactory connectionFactory(@Value("${rabbit.indirizzo}") String indirizzo) {
10     CachingConnectionFactory cachingConnectionFactory =
11         new CachingConnectionFactory(indirizzo);
12     return cachingConnectionFactory;
13 }
14 ...

```

Nella classe UploadDownloadThymeleafApplication avviene la connessione con il servizio Rabbitmq e viene creata la coda sulla quale il server scriverà, ovvero sulla coda2.

```

1 ...
2 @RabbitListener(queues = "${coda.consumatore}" )
3 public void run(Message message) throws InterruptedException,IOException {
4
5     String body=new String(message.getBody());
6     System.out.println("\t\t\t\t\t\t\t\t[received]:" +body+ "");
7     JSONObject obj= new JSONObject(body);
8     //chiamata al producer per inviare messaggi sulla coda 2
9     producer.run(obj.getString("body"),obj.getString("mittente"), true);
10    //salvataggio del messaggio sul database
11    messaggioService.addMessaggio(
12        new Messaggio(obj.getString("mittente"),obj.getString("body"),"messaggio"));
13 }
14 ...

```

Nella classe Consumer vengono letti i messaggi della coda1, vengono rigirati nella coda2 e con "messaggioService.addMessaggio" vengono memorizzati nel database.

4.3 Frontend e Backend file yaml

Anche per questi due servizi sono presenti delle immagini personalizzate, sono create in maniera simile, quindi verrà descritta la creazione di una sola di loro. Bisogna andare nella cartella del progetto frontend, eseguire "mvn clean package" per creare il file .jar nella cartella "target", rinomiarlo "client.jar", spostarlo dove è presente il dockerfile appropriato ed eseguire "docker build . -f Dockerfile_frontend -t stecol/client_spring_boot" (si sta ipotizzando che il nome del dockerfile è "Dockerfile_frontend"), in maniera analoga viene fatto per il servizio backend.

```
FROM openjdk
COPY client.jar /
CMD ["java", "-jar", "client.jar"]
```

```
FROM openjdk
COPY server.jar /
CMD ["java", "-jar", "server.jar"]
```

Nei file .yaml i servizi presentano configurazioni simili: col tag "image" si indica che i container "si basano sulle immagini personalizzate che sono state create prima; espongono la porta 8080; con il tag "depends_on" indicano che i loro container devono essere creati dopo la creazione dei servizi "mariadb" e "rabbitmq"; infine con "environment" vengono indicate le variabili d'ambiente che i container devono conoscere. "SERVER_PORT" indica su quale porta esporre il server Tomcat di Spring Boot, "SPRING_DATASOURCE_URL" indica il link per collegarsi al database MariaDB e "RABBIT_INDIRIZZO" indica l'indirizzo per collegarsi al servizio RabbitMQ. Come si può vedere dall'url e dall'indirizzo, sono stati utilizzati i nomi dei servizi invece degli indirizzi veri e propri, questo è possibile perché Kubernetes e Spring Boot riescono a risolvere i nomi, traducendoli negli effettivi indirizzi dei container.

```
...
client:
  image: 'stecol/client_spring_boot:latest'
  ports:
    - "8080:8080"
  depends_on:
    - "mariadb"
    - "rabbitmq"
  environment:
    - SERVER_PORT=8080
    - SPRING_DATASOURCE_URL=jdbc:mysql://mariadb/chat
    - RABBIT_INDIRIZZO=rabbitmq
...
```

```
...
server:
  image: 'stecol/server_spring_boot:latest'
  ports:
    - "8080:8080"
  depends_on:
    - "mariadb"
    - "rabbitmq"
```

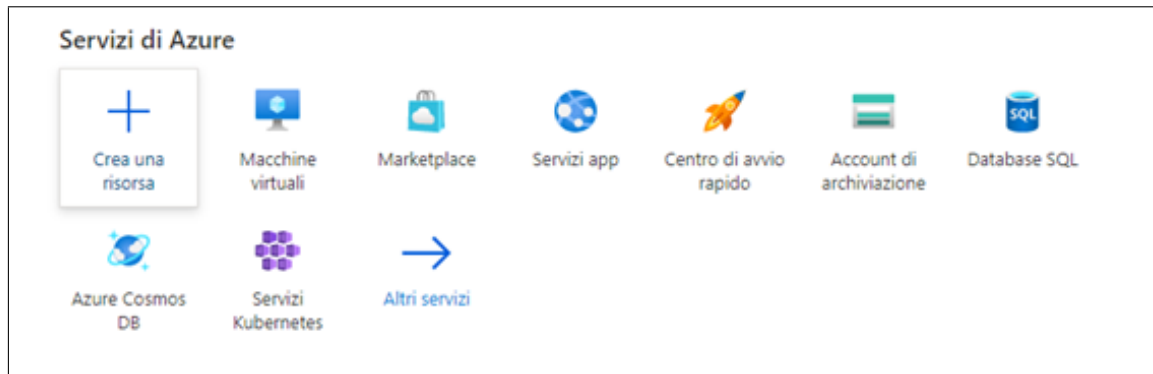
environment:

- SERVER_PORT=8080
- SPRING_DATASOURCE_URL=jdbc:mysql://mariadb/chat
- RABBIT_INDIRIZZO=rabbitmq

...

5 Creazione Macchine Virtuali Azure

In questo capitolo verrà mostrato brevemente come creare le macchine virtuali, verranno indicate solo le modifiche importanti, il resto può essere lasciato all'auto compilazione. Per la creazione delle VM di Azure bisogna cliccare su "Macchine virtuali".



Cliccare su "crea".



Nella sezione "informazioni di base" bisogna:

- scegliere un gruppo di risorse (meglio unico per tutte le macchine che si devono creare).
- mettere nome della macchina (per esempio si possono creare due macchine, un kmaster e un kworker).
- selezionare un'Area che sia uguale per tutte le VM.
- scegliere come immagine ubuntu 20.04.
- scegliere una macchina che abbia almeno 2 cpu virtuali.
- Impostare il nome della chiave ssh per collegarsi alle macchine
- Impostare il nome utente (meglio inserire lo stesso per tutte le macchine).

Dettagli del progetto

Selezionare la sottoscrizione per gestire le risorse distribuite e i costi. Usare i gruppi di risorse come le cartelle per organizzare e gestire tutte le risorse.

Sottoscrizione * ⓘ Azure for Students ▼

Gruppo di risorse * ⓘ Mine1_group ▼
[Crea nuovo](#)

Dettagli istanza

Nome della macchina virtuale * ⓘ

Area * ⓘ (Europe) West Europe ▼

Opzioni di disponibilità ⓘ La ridondanza dell'infrastruttura non è richiesta ▼

Tipo di sicurezza ⓘ Standard ▼

Immagine * ⓘ Ubuntu Server 20.04 LTS- Gen2 ▼
[Visualizza tutte le immagini](#) | [Configura generazione macchina virtuale](#)

Istanza Spot di Azure ⓘ ☐

Dimensioni * ⓘ Standard_B2s - 2 cpu virtuali, 4 GiB di memoria (35,04 USD/mese) ⓘ ▼
[Visualizza tutte le dimensioni](#)

Nella sezione "rete" bisogna:

- impostare una rete virtuale esistente o crearne una nuova (si dovrebbe auto compilare con un nome simile al gruppo di risorse scelto prima).
- scegliere la subnet che deve essere la stessa per tutte le macchine.
- inserire un ip pubblico statico per collegarsi con ssh.

Interfaccia di rete

Quando si crea una macchina virtuale, viene creata automaticamente un'interfaccia di rete.

Rete virtuale * ⓘ Mine1-vnet ▼
[Crea nuovo](#)

Subnet * ⓘ default (10.1.1.0/24) ▼
[Gestisci configurazione subnet](#)

IP pubblico ⓘ (nuovo) ip-macchina ▼
[Crea nuovo](#)

Gruppo di sicurezza di rete della scheda di interfaccia di rete ⓘ ☐ Nessuno ☒ Basic ☐ Avanzate

Porte in ingresso pubbliche * ⓘ ☐ Nessuno ☒ Consenti porte selezionate

Selezionare le porte in ingresso * SSH (22) ▼

⚠ A tutti gli indirizzi IP sarà consentito accedere alla macchina virtuale.
 Questa opzione è consigliata solo per il test. Usare i controlli avanzati nella scheda Rete per creare regole per limitare il traffico in ingresso a indirizzi IP noti.

Elimina l'indirizzo IP pubblico e la scheda di interfaccia di rete quando viene eliminata la macchina virtuale ⓘ ☐

Per creare le macchine infine basta cliccare su "rivedi e crea" ed aspettare qualche minuto per avere le VM pronte.

6 Configurare le VM con Kubernetes e far partire i deployment

Per accedere alle macchine virtuali si utilizza il comando "ssh -i chiave.pem user@ip" dove chiave.pem è la chiave scaricata dal portale di azure per accedere alle macchine, user è l'utente che stato definita nei passaggi sopra per creare le macchine virtuali e ip è l'ip pubblico della macchina. Per inviare file alle macchine virtuali si utilizza il comando "scp -i chiave.pem nome_file user@ip:". I file utilizzati per il setup sono presenti nell'archivio github del progetto.

Una volta che si è riusciti ad avere accesso alle macchine virtuali si può lanciare lo script "configurazione.sh" che presenta comandi per installare Docker e Kubernetes all'interno delle macchine virtuali. Bisogna solo modificare gli ip all'inizio del file.

Completata l'inizializzazione del master, bisogna inizializzare il cluster con il seguente comando. Per "IP_RETE_PRIVATA" si intende l'ip della subnet che è stato assegnato all'VM.

```
sudo kubeadm init --apiserver-advertise-address=IP_RETE_PRIVATA \
--pod-network-cidr=192.168.0.0/16
```

Eseguire i comandi che l'output del comando precedente ci suggerisce di usare.

```
mkdir -p $HOME/.kube
sudo rm -rf $HOME/.kube/config
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Creare la rete calico.

```
kubectrl create -f calico.yaml
```

Aggiungere un settaggio alla rete e riavviare i coredns per far funzionare la risoluzione dei nomi dei servizi.

```
kubectrl set env daemonset/calico-node -n kube-system \
IP_AUTODETECTION_METHOD=interface=eth0
kubectrl rollout restart deployments -n kube-system coredns
```

Sportarci sul kworker dove dobbiamo eseguire il comando fornito in output dal comando precedente "kubeadm init" che ha una forma di questo tipo.

```
kubeadm join 10.1.1.10:6443 --token 1fuuic.sdy3dyqxhn83ss4r \
--discovery-token-ca-cert-hash sha256:1dfb1c6
```

Bisogna prima installare il tool "Kompose", che permette di convertire un file adatto per docker-compose in un file per Kubernetes.

```
wget https://github.com/kubernetes/kompose/releases/download/v1.26.1/kompose_amd64.deb
sudo apt install ./kompose_1.26.1_amd64.deb
```

Si lancia il seguente comando che creerà un insieme di file .yaml.

```
kompose convert -f docker-compose.yaml
```

Kompose crea quasi tutti i file in maniera perfetta, ma ha delle difficoltà per quanto riguarda i file dei volumi persistenti, quindi a questo punto utilizzare i file .yaml relativi ai volumi, presenti sul repository github. Ed infine si possono lanciare i deployment.

```
kubectl apply -f .
```

Nel caso in cui si possano presentare problemi perche il frontend e il backend non riescono a risolvere i nomi dei servizi, utilizzare il seguente comando per eseguire il restart dei servizi: "kubectl rollout restart deployments client kubectl rollout restart deployments server". Adesso l'applicazione risulta essere funzionante ma non è ancora raggiungibile dall'esterno; quindi, eseguiamo il seguente comando che permette all'utente di raggiungere l'interfaccia web con l'ip del kmaster alla porta 8080.

```
kubectl port-forward --address IP_RETE_PRIVATA service/client 8080:8080
```

Questo è il risultato.

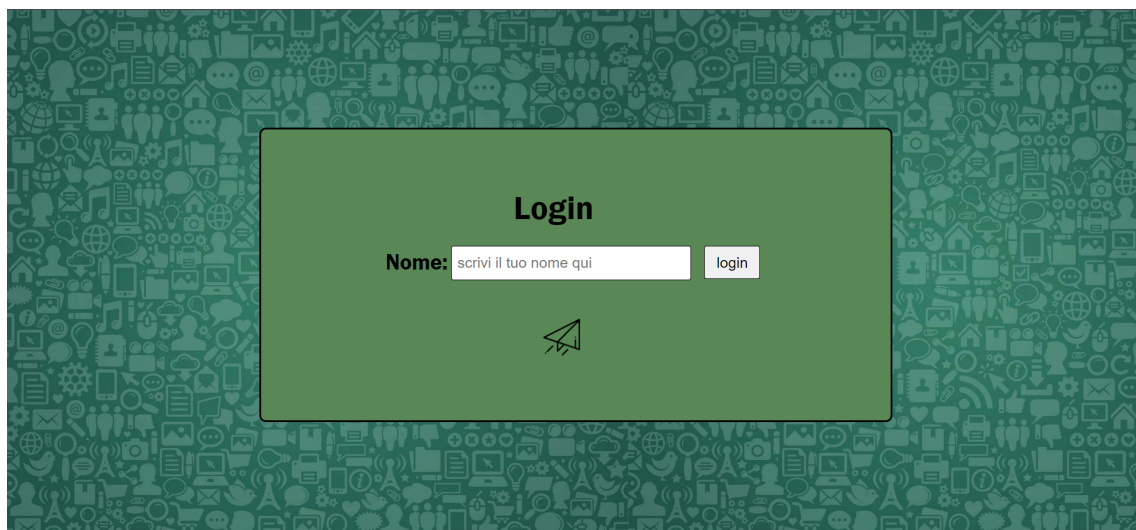


Figura 9: Login



Figura 10: Chat