
# Relazione progetto di Ingegneria dei Sistemi Distribuiti

## Creazione di una chat per lo scambio di messaggi e invio di file

Stefano Colombo
Matricola 1000024706

# Indice

# 1 Introduzione

L'elaborato prodotto mostra lo sviluppo di una chat asincrona che permette l'invio di messaggi e file tra due utenti finali. 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 Ubuntu, configurati appositamente per eseguire applicazioni Spring Boot, dove è implementata la logica della chat.

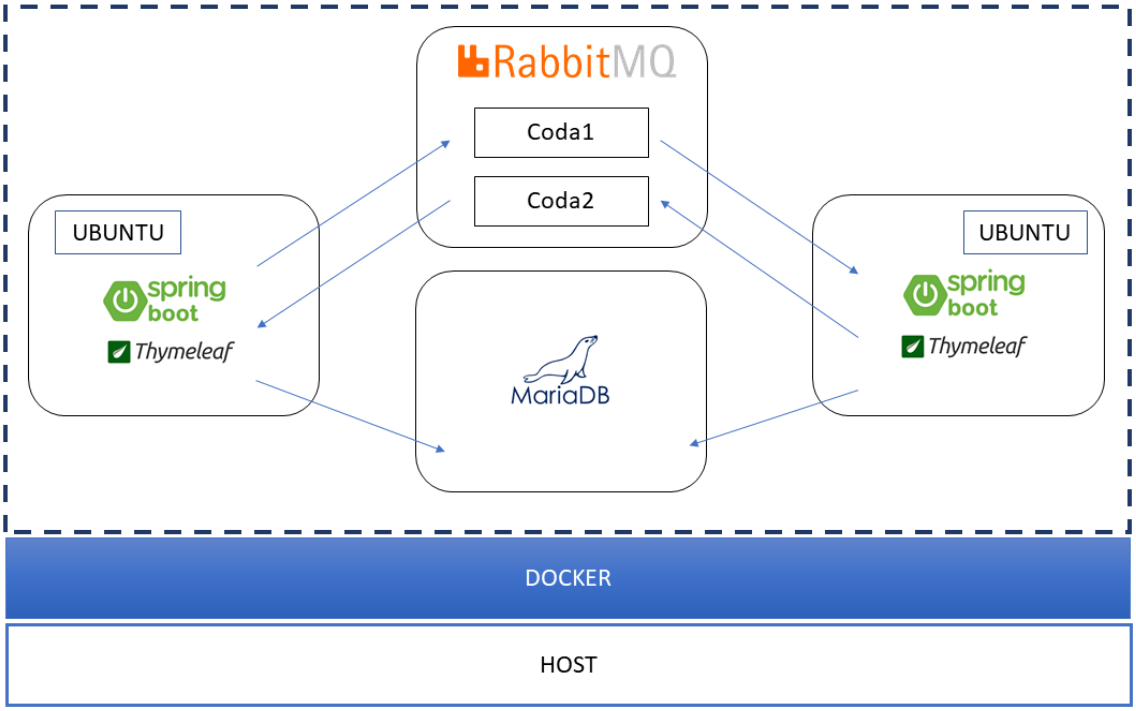


Figura 1: Struttura progetto

Per eseguire i container verrà utilizzato il tool docker-compose che permette di definire ed eseguire applicazioni Docker multi-container. La configurazione di ogni container viene definita in un file YAML (le cui parti verranno mostrate nei capitoli successivi), che insieme al tool docker-compose permette di eseguire e stoppare container con un semplice comando.

# 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 o un file, se ad essere inviato è un messaggio, viene memorizzato il testo, se si dovesse trattare di un file, vengono memorizzati il nome, l'estensione e il link (utlizzato dall'interfaccia utente per permettere il download dei file), in entrambi i casi vengono memorizzati il mittente e il tipo di messaggio (file o 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 | |
| estensione | varchar(255) | YES | | NULL | |
| link | varchar(255) | YES | | NULL | |
| mittente | varchar(255) | YES | | NULL | |
| nome | varchar(255) | YES | | NULL | |
| tipo | varchar(255) | YES | | NULL | |

Figura 2: Tabella database

Nel file docker-compose.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 4002; 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/
    environment:
      – 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 Ubuntu di comunicare.

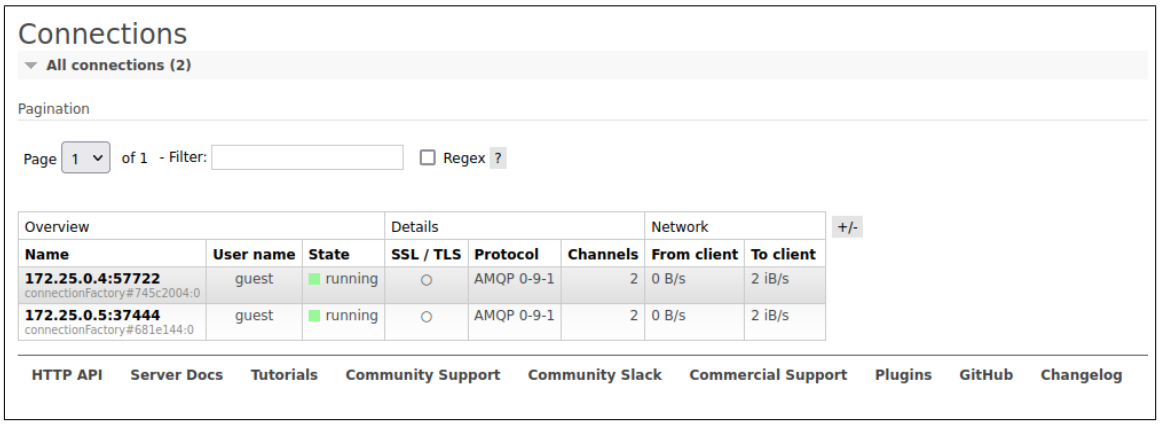


Figura 3: Connessioni al servizio RabbitMQ

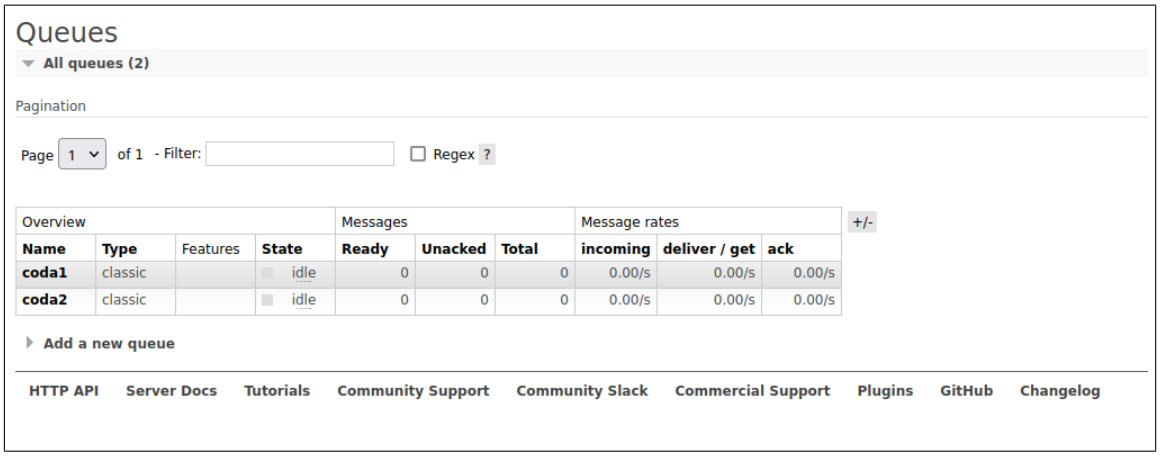


Figura 4: Code create

Nel file docker-compose.yml il servizio RabbitMQ presenta le seguenti configurazioni: si basa su l'immagine "rabbitmq:management-alpine" ed espone le porte 5672 e 15672. La porta 5672 serve ai container per collegarsi al servizio mentre 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 per monitorare l'utilizzo del container da parte delle connessioni in entrata. Le immagini sopra mostrano le connessioni a cui il servizio è collegato e le code create.

```
...
rabbitmq:
    image: 'rabbitmq:management-alpine'
    ports:
      - '5672:5672'
      - '15672:15672'
...
```

# 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 verrà mostrato, prima la struttura e parte del codice utilizzato per sviluppare l'applicazione, contente la logica della chat, e successivamente le immagini create ed utilizzate per far eseguire l'applicazione all'interno dei container Ubuntu.

## 4.1 Struttura

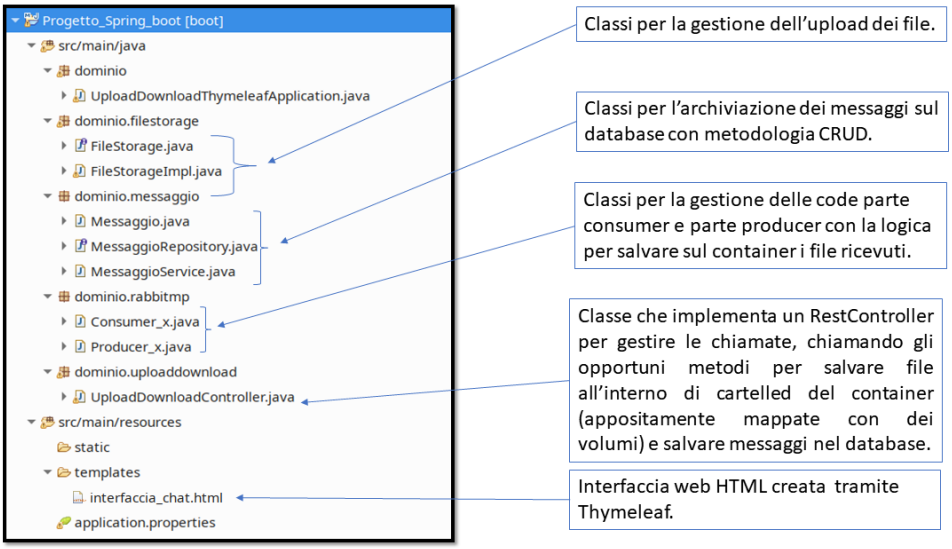


Figura 5: Struttura app

Le classi che compongono l'applicazione sono suddivise in package in base al loro ruolo. Le classi col prefisso "FIleStorage" si occupano dell'upload dei file, che verranno inviati successivamente tramite le code RabbitMQ, per fare ciò il file viene prima caricato in una cartella interna al container per poi essere letto e inviato dalle classi che si occupano della gestione dell'invio dei dati tramite le code. 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. Le classi del package "rabbitmq" permettono di inviare e ricevere dati dalle code RabbitMQ, se si tratta di file, si occupano di leggerli per l'invio, o di memorizzarli per la ricezione. La Classe UploadDownloadController svolge il ruoto 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 MessaggeService e di memorizzare/inviare file tramite le opportune classi dei package "filestorage"

e "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. Infine il file "interfaccia_chat.html" è l'interfaccia web creata tramite il tool Thymeleaf che permette di manipolare i dati ricevuti dal RestController per creare una pagina HTML.

## 4.2 Producer

La classe Producer svolge la funzione di invio dei dati nella coda RabbitMQ, un messaggio può avere una dimensione massima che si aggira intorno ai 130MB, oltre quella cifra, la coda rifiuta di inviare il messaggio, Per questo motivo quando bisogna inviare un file, qualunque sia la dimensione, occorre inviare una porzione di dati alla volta e, quando le diverse porzioni giungono a destinazione bisogna riassemblarle in modo tale che si ottenga di nuovo il file originale

```
...
@Autowired
private Queue queue;

@Autowired
private RabbitTemplate rabbit;

public void run(String mess,String temp, boolean messaggio) {
        if(messaggio==false)inviaFile(mess,temp);
        else rabbit.convertAndSend(queue.getName(), mess);
}
...
```

Tramite le annotazioni @Autowired si ottengono le istanze di Queue e RabbitTamplete i cui bean sono stati definiti nei file di configurazione. Il metodo "run" distingue i messaggi dai file: i messaggi vengono inviati direttamente nella coda mentre se si dovesse trattare di un file viene richiamata la funzione "inviaFile", il cui codice è mostrato sotto.

```
...
public void inviaFile(String mess,String temp){
        byte[] fileData ;
        MessageProperties mp=new MessageProperties();
        mp.setContentType(MessageProperties.CONTENT_TYPE_BYTES);

        FileInputStream is = null;

        //invio del nome e del tipo di estensione
        String parti[]=temp.split("\\.");
        String tempNome="";
        for(int i=0;i<parti.length-1;i++) {
                tempNome+=(parti[i]+".");
        }
        String nome="nome:"+tempNome.substring(0,tempNome.length()-1);

```

```
            System.out.println(parti[0]);
            if(parti.length>1)
            rabbit.convertAndSend(queue.getName(), "estensione:"+parti[parti.length-1]);
            rabbit.convertAndSend(queue.getName(),nome );

            try {
                is = new FileInputStream(mess);
                byte[] buffer= new byte[1024];
                byte[] arraypiccolo;
                int pos=0;
                rabbit.convertAndSend(queue.getName(), "inizio_invio_file");
                while((pos=is.read(buffer))!=-1) {

                    if(pos<1024) {
                            arraypiccolo = Arrays.copyOf(buffer,pos);
                            rabbit.send(queue.getName(),new Message(arraypiccolo,mp));
                    }
                    else rabbit.send(queue.getName(),new Message(buffer,mp));
                }
                rabbit.convertAndSend(queue.getName(), "fine");
              }catch (IOException e) {
                    System.out.println("Errore conversione in byte"+e);
                    e.printStackTrace();
              }finally {
                    try {
                            is.close();
                    } catch (IOException e) {
                            e.printStackTrace();
                    }
              }
}
...
```

Prima di inviare il file vero e proprio il producer invia messaggi riguardanti l'estensione del file, il nome del file e un messaggio "inizio_invio_file" che notifica al destinatario che l'invio del contenuto del file sta per iniziare. Come è stato accennato prima, i messaggi hanno una dimensione massima; quindi, si utilizza un ciclo while per leggere, dal file, 1024 byte, per poi inviarli alla coda; i file che si vogliono inviare, molto probabilmente, non hanno una dimensione multipla di 1024 byte; quindi, l'ultimo pezzo di file viene inviato con una dimensione appropriata, se così non fosse i byte superflui potrebbero corromperlo. Terminato il ciclo while si invia un messaggio che indica la fine della trasmissione del file, questo notificherà al consumer il momento adatto per memorizzare il record nel database, visto che l'invio di un file verrà memorizzato, solamente una volta che il consumer avrà terminato la sua ricezione.

## 4.3 Consumer

La classe Consumer permette di ricevere i dati dalla coda dei messaggi, non rimane continuamente in ascolto con cicli infiniti ma utilizza il meccanismo di callback nel quale sarà la coda ad informare l'applicazione della presenza di messaggi da poter ricevere.

```
...

@Component
public class Consumer{

        @Autowired
        MessaggioService messaggioService;

        @Autowired
        public Consumer_x(@Value("${custom.percorsoFileRicevuti}")
        String percorsoFileRicevuti,@Value("${custom.destinatario}") String destinatario) {
            this.percorsoFileRicevuti=percorsoFileRicevuti;
            this.destinatario=destinatario;
        }
...
```

L'annotazione @Component consente a Spring di rilevare automaticamente i bean personalizzati, in altre parole, senza dover scrivere alcun codice esplicito, Spring scansiona l'applicazione e per le classi annotate con @Component crea un'istanza e inietta in esse le dipendenze specificate. L'annotazione @Autowired permette di ottenere l'istanza di MessaggioService per invocare i metodi necessari per la memorizzazione dei messaggi all'interno del database MariaDB e tramite la notazione "${...}" prende le variabili definite nel file application.properties o le variabili d'ambiente che vengono definite durante la creazione del container con docker-compose.

```
...
@RabbitListener(queues = "${coda.consumatore}")
public void run(Message message) throws InterruptedException,IOException {
        FileOutputStream outputStream=null;
        //file in arrivo
        if(new String(message.getBody()).split(":")[0].equals("estensione")) {
          estensione=getInfo(message);//tipo del file
        }
        else if(new String(message.getBody()).split(":")[0].equals("nome")) {
          nomefile=getInfo(message);//nome del file
        }
        else if(new String(message.getBody()).equals("inizio_invio_file")) {
          inizio=true;
          outputStream = new FileOutputStream(pathFileRicevuti+nomefile+"."+estensione);
        }
        else if(!nomefile.equals("")){
          if(inizio==true) {
           if(!(new String(message.getBody()).equals("fine")) {
                try {
```

```
                        String temp=pathFileRicevuti+nomefile+"."+estensione;
                        outputStream = new FileOutputStream(tmp,true);
                        outputStream.write(message.getBody());
                    }catch(IOException e){
                        System.out.println("Errore:"+e);
                    }
                }
                else{
                    String nome="";
                    if(estensione.equals(""))nome=nomefile;
                    else nome= nomefile+"."+estensione;

                    Message m=new
                    Messaggio(destinatario,"vuoto","file",nome,estensione,"url"));
                   messaggioService.addMessaggio(m)

                    estensione="";
                    nomefile="";
                    grande=false;
                   }
              }
            }
            //messaggio in arrivo
            else {
             String body=new String(message.getBody());
             //savaltaggio del messaggio sul database
             Messaggio m=new Messaggio(destinatario,body,"messaggio","","","")
             messaggioService.addMessaggio();
            }
}
...
```

La notazione @RabbitListener permette di definire la coda da cui prelevare i messaggi, in questo caso il nome della coda viene definito da una variabile nel file YAML, dato in input a docker-compose. I primi due if servono per definire nome ed estensione del file che si aspetta di ricevere, il terzo if serve per impostare la variabile booleana "inizio" a true, per indicare che dal prossimo messaggio verranno inviate porzioni di file e tramite la libreria "FileOutputStream", viene creato un file vuoto, nella cartella definita da "pathfileRicevuti", sul file system del container; una volta definiti nome, estensione e path del file, l'applicazione riceve array di byte che scriverà sul file creato in precedenza che adesso viene aperto in modalità append (settando a "true" il secondo attributo del costruttore "FileOutputStream"); quando viene ricevuto il messaggio "fine" si sta indicando che il produttore ha inviato tutte le porzioni del file e adesso il consumatore memorizza un record sul database indicando il destinatario, il nome e l'estensione del file che ha appena ricevuto. Nel caso in cui si riceve un normale messaggio, il consumatore memorizza nel database il destinatario e il contenuto del messaggio.

## 4.4 RestController

Questa classe ci permette di mappare le richieste Rest provenienti dall'interfaccia utente creata tramite thymeleaf con opportuni metodi che permetto di memorizzare messaggi nel database e immagazzinare file nel file system del container.

```
...
    @GetMapping("/")
     public String listaMessaggi(Model model) throws IOException {

                //preleviamo i messaggi dal database
                List<Messaggio> list=messaggioService.getAllMessaggi();

                //generiamo il link dei file che abbiamo
                for(Messaggio m:list) {
                        String url = MvcUriComponentsBuilder.fromMethodName(
                                UploadDownloadController.class,
                            "downloadFile",
                            m.getNome() ).build().toString();
                                m.setLink(url);
                }
                        model.addAttribute("messaggi",list);
            return "interfaccia_chat";
      }
```

Quando avviene la connessione al servizio tramite l'interfaccia utente, il primo metodo che viene richiamato è "listaMessaggi", questo metodo richiede al database tutto lo storico dei messaggi, genera il link per i file presenti nel container, inviati e ricevuti durante la comunicazione, e ritorna questi dati in modo tale che Thymeleaf li possa manipolare e gestire per fornire un'interfaccia web all'utente.

```
...
@PostMapping("/messaggio")
public String InvioMessaggio(Messaggio e, Model model) {
            if (e.getId() != null) {
                        model.addAttribute("Messaggio",null);
            }
            //mando il messaggio al producer
            producer.run(e.getContenuto(),"vuoto", true);
            return "redirect:";
}
...
```

Quando viene inviato un semplice messaggio, questo metodo viene richiamato e non fa altro che inviare il contenuto del messaggio al producer per poi ritornare sulla pagina principale. "redirect" permette di ridirezionare una richiesta ad un'altra pagina, in questo caso, non inserendo nulla dopo i due punti, si indica di voler essere ridirezionati all'URL "/".

```
...
@PostMapping("/")
public String InvioFile(@RequestParam("files") MultipartFile[] files, Model model) {
        try {
        Arrays.asList(files).stream()
            .map(file -> {
                        fileStorage.store(file);
                        producer.run(fileStorage.getPercorso()+
                        file.getOriginalFilename(),\
                        file.getOriginalFilename(), false);
                return file.getOriginalFilename();
                                }).collect(Collectors.toList());
                model.addAttribute("message", "File caricato con successo!");
        } catch (Exception ex) {
                model.addAttribute("message", "Fail! errore nel caricamento");
        }
    return "redirect:";
}
...
```

Il form presente nell'interfaccia web, permette di inviare più file alla volta; ogni file verrà immagazzinato nel file system mittente così che il producer può leggerli e successivamente inviarli.

```
...
//Download Files
@GetMapping("/{filename}")
  public ResponseEntity<Resource> downloadFile(@PathVariable String filename) {
        Resource file = fileStorage.loadFile(filename);
        return ResponseEntity.ok()
          .header(HttpHeaders.CONTENT_DISPOSITION,
          "attachment;filename=\"" + file.getFilename() + "\"") .body(file);
}
..
```

Questo metodo permette all'utente di scaricare file dalla chat cliccando sul link che gli viene mostrato nell'interfaccia. Questo è possibile grazie all'intestazione "CONTENT_DISPOSITION", accompagnata da un allegato, che vengono ritornati come risposta al metodo.

## 4.5 Configurazione

È interessante notare come la configurazione definita nel file "application.properties", grazie all'utilizzo di docker-compose, risulta essere diminuita e semplificata. Le variabili come il nome delle code, il ruolo dell'applicazione nella chat e le porte, che devono essere utilizzate dal server Tomcat, possono essere definite come variabili d'ambiente nel file YAML (infatti nel file sottostante sono commentate) e l'indirizzo dei container a cui l'applicazione deve collegarsi è sostituito dal nome dei servizi definiti sul file YAML, questo è molto utile soprattutto perché evita il doversi affidare ad indirizzi che possono cambiare nel tempo, rendendo impossibile la comunicazione. Le configurazioni rimaste nel file sono le seguenti: quelle relative all'upload del file, indicando le dimensioni massime (1000MB) e la soglia dopo la quale il file viene memorizzato sul disco (1MB); le cartelle del container dove immagazzinare i file ricevuti e caricati e i dettagli per le connessioni ai container MariaDB e RabbitMQ.

```
#configurazione multipart file
spring.servlet.multipart.enabled=true
spring.servlet.multipart.file-size-threshold=1MB
spring.servlet.multipart.max-file-size=1000MB
spring.servlet.multipart.max-request-size=1000MB

#dove memoriazzare i file caricati e scaricati
custom.percorsoFileRicevuti=/home/download/
custom.percorsoFileCaricati=/home/upload/

#connessione database
#PRIMA: spring.datasource.url=jdbc:mysql://172.17.0.1:4002/chat
spring.datasource.url=jdbc:mysql://mariadb/chat
spring.datasource.username=stefano
spring.datasource.password=ste
spring.jpa.hibernate.ddl-auto = update

#indirizzo del brocker dei messaggi
#PRIMA: rabbit.indirizzo=172.22.0.1
rabbit.indirizzo=rabbitmq

#servono a thymeleaf per capire chi sta parlando nella chat
#custom.mittente=partecipante1
#custom.destinatario=partecipante2

#il nome delle code da creare
#coda.produttore=coda1
#coda.consumatore=coda2

#server.port=8081
```

## 4.6 Creazione Container

Per creare il container che possa far eseguire l'app spring boot, prima bisogna creare un'immagine "base". Si inizia creando un immagine "base_ubuntu" con il Dockerfile sottostante usando il comando "docker build . -t base_ubuntu"

```
FROM ubuntu:latest
RUN apt−get update −y
RUN apt−get upgrade −y
RUN mkdir /home/download
RUN mkdir /home/upload
```

Adesso bisogna attaccarsi al container in esecuzione, installare openjdk e poi creare un'immagine dal container. Per fare ciò si esegue il container con il comando "docker run -it –name base base_ubuntu" dove i flag "-it" permettono di attaccarsi allo standard output ed input del container; all'interno del container si installa openjdk con il comando "apt-get install default-jdk" (durante l'istallazione bisognerà digitare: y, 8 e 41 in seguito alle domande che compariranno) ed infine si crea l'immagine chiamata "base_openjdk" ,dal container in esecuzione, utilizzando il comando "docker commit base base_openjdk".

Dopo aver ottenuto l'immagine di base, necessaria, si crea l'immagine, del container, utilizzata nell'elaborato. Con il Dockerfile sottostante e il comando "docker build . -t container_spring_boot" si ottiene l'immagine "container_spring_boot". Il file "progetto.jar" è ottenuto dopo aver utilizzato il comando "mvn clean package" nella cartella del progetto spring boot (il file .jar verrà creato nella cartella target del progetto).

```
FROM base_openjdk
COPY progetto.jar /
CMD ["java","−jar","progetto.jar"]
```

Una volta ottenuto l'immagine interessata si può eseguire il comando docker-compose.

```
...
parte1:
    image: container_spring_boot:latest
    ports:
      − "8080:8080"
    volumes:
      − download−parte1:/home
    environment:
      − CUSTOM_MITTENTE=partecipante1
      − CUSTOM_DESTINATARIO=partecipante2
      − CODA_PRODUTTORE=coda1
      − CODA_CONSUMATORE=coda2
    depends_on:
      − "mariadb"
      − "rabbitmq"
...
```

Nel file docker-compose.yml il container parte1 presenta le seguenti configurazioni: con l'attributo "image" si indica di utilizzare l'immagine di base "container_spring_boot";

con "ports" il container espone la porta 8080, grazie alla quale l'utente può utilizzare l'interfaccia grafica; con "volumes" viene mappato il contenuto della cartella "/home" all'interno del volume di nome "download-parte1",dentro la cartella "home" sono presenti le cartelle "download" e "upload", questa configurazione permettere ai file ricevuti e caricati di sopravvivere tra un'esecuzione e l'altra dei container; tramite "enviroment" si definiscono le variabili d'ambiente, la prima coppia indica i ruoli che quel servizio ha nella chat e la seconda indica il nome delle code che devono essere utilizzate per inviare e ricevere dati; e infine "depends_on" indica che l'avvio del container verrà posticipato fino a quando i servizi "mariadb" e "rabbitmq" non saranno attivi, se così non fosse, l'applicazione ritorna errore e il container viene stoppato. La configurazione del servizio "parte2" è molto simile, cambiano solo i ruoli della chat, le code usate e la porta esposta.

```
...
parte2:
    image: container_spring_boot:latest
    ports:
      - "8081:8080"
    volumes:
      - download-parte2:/home
    environment:
      - CUSTOM_MITTENTE=partecipante2
      - CUSTOM_DESTINATARIO=partecipante1
      - CODA_PRODUTTORE=coda2
      - CODA_CONSUMATORE=coda1
    depends_on:
      - "mariadb"
      - "rabbitmq"
...
```

# 5 Conclusione

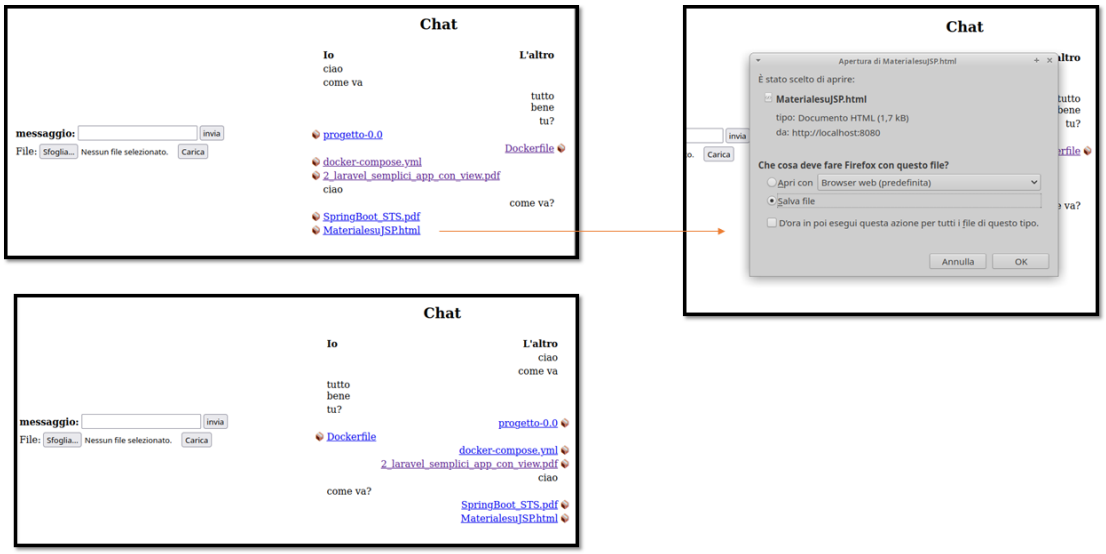


Figura 6: Interfaccia utente

Nell'elaborato trattato precedentemente si è visto lo sviluppo del progetto di una chat che permette all'utente finale di poter inviare e ricevere messaggi e file, progettata con un'architettura Docker multi-container. Questo è stato possibili grazie alla presenza di un container RabbitMQ che ha permesso la gestione delle code per l'invio dei dati; un container MariaDB che ha permesso, tramite l'utilizzo di volumi, di memorizzare lo storico della chat in maniera persistente; e dei container Ubuntu configurati appositamente per eseguire applicazioni Spring Boot che hanno permesso la gestione della logica della chat, tra le quali la possibilità di inviare file di grandi dimensioni, di gestire la memorizzazione dei file sul file system dei container, di comunicare col database e di fornire l'interfaccia, creata tramite Thymleaf, mostrata sopra, che offre la possibilità all'utente non solo di inviare dati ma anche di poter eseguire il download dei file, semplicemente cliccando sul loro nome mostrato nella chat. Come ogni software esistente, anche a questa applicazione possono essere aggiunte nuove componenti e nuove funzionalità, alcune idee potrebbero includere la presenza di un interfaccia che permetta all'utente di compiere più operazioni, come la cancellazione dei messaggi o la visualizzazione di immagini, si potrebbe pensare anche ad una componente di caching per i messaggi meno recenti o anche la possibilità di inserire un meccanismo di autenticazione per certificare l'entità dell'utente che utilizza la chat.