

---

# *Progetto Ingegneria dei sistemi distribuiti*

---

*Giovanni Martucci*

## Table of Contents

Obiettivo .....	3
1. Cenni sulla teoria .....	3
1.1 RabbitMQ .....	3
1.2 Reference Monitor .....	4
1.3 Docker.....	4
1.4 Angular .....	5
1.5 Nodejs .....	5
1.6 Firebase .....	5
2. Specifiche tecniche .....	6
3. Modalità d’esecuzione .....	6
3.1 Sviluppo comunicazione RabbitMQ.....	8
3.1.1 Exchange direct – Filter - Routing.....	9
3.1.2 RPC.....	9
3.1.3 Gestione comunicazione con Timeout.....	10
3.2 Diagrammi UML e diagramma di flusso .....	11
3.3 Dockerizzazione.....	13
4. Test e Risultati .....	16
5. Usage.....	18
Conclusione.....	19
Reference .....	20
Repository GitHub .....	20

## Obiettivo

Questo progetto si prepone l'obiettivo di sviluppare una web-app, contenente una dashboard nel quale sono presenti diversi servizi disponibili. Nello specifico, dall'interfaccia web è possibile utilizzare due dei tre servizi offerti dalla piattaforma. Al momento dell'iscrizione un utente può iscriversi ad un massimo di due servizi, in questa maniera sarà possibile avere un riscontro della corretta ed effettiva gestione delle autorizzazioni. Il flusso dell'applicazione prosegue come segue: i client, dopo aver effettuato un signup/signin, inviano una richiesta al back-end (che interpreterà il ruolo di Reference Monitor) per un determinato servizio che vogliono utilizzare. Il back-end, controllerà i permessi dell'utente in questione, e successivamente in caso di esito positivo, interrogherà il micro-servizio selezionato, utilizzando RabbitMQ.

## 1. Cenni sulla teoria

### 1.1 RabbitMQ

RabbitMQ [1] è un broker di messaggi e un server di code che può essere usato da applicazioni per condividere dati. RabbitMQ implementa lo standard aperto AMQP (Advanced Message Queuing Protocol). Esso ha il ruolo di broker fra l'app e il server con cui vuole dialogare l'app. I Produttori creano messaggi e li pubblicano ad un servizio broker (RabbitMQ). Il messaggio ha due parti: payload, e label. Il payload è quel che si vuole trasmettere. La label descrive il payload, così che RabbitMQ determina chi dovrà avere una copia del messaggio. I Consumatori si attaccano ad un broker e si sottoscrivono ad una coda. Quando un messaggio arriva in una coda, RabbitMQ lo manda a uno dei consumatori in ascolto. La pubblicazione di un messaggio, la sottoscrizione ad una coda o la ricezione di un messaggio, sono tutte operazioni fatte su un canale. Per poter mandare un messaggio si hanno tre parti: scambi (exchange), code (queue) e connessioni (binding). Il produttore non invia i messaggi direttamente alla coda, invece li invia ad uno scambio (exchange). Un exchange da un lato riceve messaggi dai produttori e dall'altro lato li manda alle code. Ci sono diversi tipi di exchange: direct, topic, headers, e fanout.

## 1.2 Reference Monitor

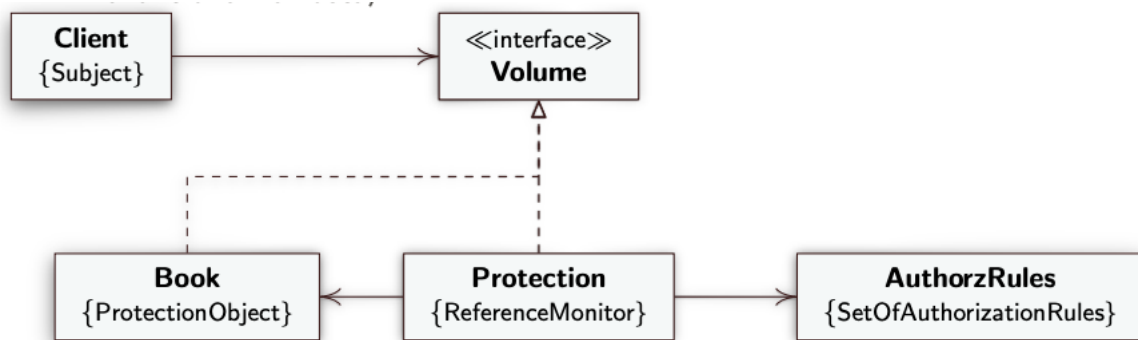


Figura 2 – Design pattern “Reference Monitor”

Il Reference Monitor [2] è un pattern largamente utilizzato per applicazioni che richiedono un controllo delle autorizzazioni. Nello specifico come mostra la figura 2 vi sono diversi ruoli da ricoprire:

- **ProtectionObject**: rappresenta l’oggetto da proteggere
- **Subject**: è il client che richiede di accedere a/ai **ProtectionObject**
- **ReferenceMonitor**: è il componente che intercetta la richiesta e cerca fra le regole di autorizzazione una regola che può essere usata, in base alla richiesta.
- **SetOfAuthorizationRules**: è un insieme di regole di autorizzazione, salvate opportunamente su un database.

## 1.3 Docker

Docker [3] è una piattaforma software che permette di creare, testare e distribuire applicazioni con la massima rapidità. Docker confeziona il software in unità chiamate container che offrono tutto il necessario per la loro corretta esecuzione, incluse librerie, strumenti di sistema, codice e runtime.



Figura 3 – Docker.

## 1.4 Angular

Angular [4] è un framework di sviluppo per realizzare applicazioni desktop, web application e mobile application. Permette la realizzazione di applicazioni complesse, sia per desktop che per web o mobile. Utilizzato per lo sviluppo front-end.



Figura 4 – Angular.

## 1.5 Nodejs

Node.js [5] è un runtime system open source multiplatforma orientato agli eventi per l'esecuzione di codice JavaScript. Node.js consente di utilizzare JavaScript per scrivere codice da eseguire lato server, ad esempio per la produzione del contenuto delle pagine web dinamiche prima che la pagina venga inviata al Browser dell'utente.

Il framework Express consente di creare potenti API di routing e di impostare middleware per rispondere alle richieste HTTP, fornendo semplici meccanismi di debugging e una rapida integrazione con vari motori di templating. Qui impiegato come server Nodejs.



Figura 5 – Nodejs.

## 1.6 Firebase

Firebase [6] è una piattaforma serverless per lo sviluppo di applicazioni mobili e web. Esso sfrutta l'infrastruttura di Google e il suo cloud per fornire una suite di strumenti per scrivere, analizzare e mantenere applicazioni cross-platform. Firebase offre infatti funzionalità come analisi statistiche dei dati, database (usando strutture noSQL), messaggistica, autenticazione, segnalazione di arresti anomali, algoritmi di machine learning per la gestione di applicazioni web, iOS e Android.



Figura 6 – Firebase.

## 2. Specifiche tecniche

Di seguito vengono elencate tutte le specifiche tecniche utilizzate per questo progetto:

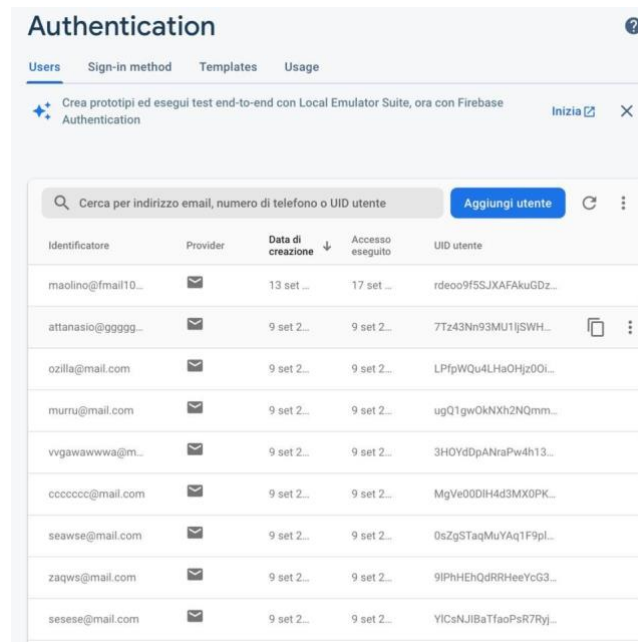
- Distro: Ubuntu 20.4, MacOS Monterey;
- Linguaggio di programmazione: TypeScript, JavaScript;
- Framework utilizzati: Angular, Nodejs, Express, Docker, RabbitMQ, Firebase, Bootstrap;
- Virtualizzazione: VMWare Fusion;

## 3. Modalità d'esecuzione

Inizialmente ci si è occupati della riprogettazione della web-app sostituendo i servizi offerti. Precedentemente la web-app fungeva da dashboard di controllo per diversi valori come la temperatura, l'umidità, ecc. I nuovi servizi offerti utilizzano API esterne intercettate sul web. Nello specifico l'utente ha a disposizione un micro-servizio dedicato al meteo, un servizio di generazione numeri (simulando il lancio di un dado a 6 facce) e infine un ricettario di cucina che in base all'elemento inserito come ingrediente principale fornisce una ricetta.

Nello specifico è stato utilizzato Angular v.13 per lo sviluppo della parte front-end e Node.js v.14 per la parte back-end e micro-servizi.

L'autenticazione dell'utente viene gestita tramite il componente login offerto dal framework Firebase.



The screenshot shows the Firebase Authentication console. At the top, there's a header with 'Authentication' and a help icon. Below it are tabs for 'Users', 'Sign-in method', 'Templates', and 'Usage'. A message bar says 'Crea prototipi ed esegui test end-to-end con Local Emulator Suite, ora con Firebase Authentication' with an 'Inizia' button. Below the message is a search bar with the placeholder 'Cerca per indirizzo email, numero di telefono o UID utente' and an 'Aggiungi utente' button. The main content is a table of users.

Identificatore	Provider	Data di creazione ↓	Accesso eseguito	UID utente
maolino@fmail10...	📧	13 set ...	17 set ...	rdeo09fSSJXAFAkuGDz...
attanasio@ggggg...	📧	9 set 2...	9 set 2...	7Tz43Nn93MU1jSWH...
ozilla@mail.com	📧	9 set 2...	9 set 2...	LPfpWQu4LHaOHjz0Oi...
murru@mail.com	📧	9 set 2...	9 set 2...	ugQ1gwOkNXh2NQmm...
vvgawawwwa@m...	📧	9 set 2...	9 set 2...	3HOYdDpANraPw4h13...
ccccc@mail.com	📧	9 set 2...	9 set 2...	MgVe00DIH4d3MX0PK...
seawse@mail.com	📧	9 set 2...	9 set 2...	0sZgSTaqMuYAq1F9pl...
zaqws@mail.com	📧	9 set 2...	9 set 2...	9lPhEHQdRRHeeYcG3...
sesese@mail.com	📧	9 set 2...	9 set 2...	YICsNJlBaTfaoPsR7Ry...

Figura 7 – Authentication Firebase

Il software segue il design pattern Reference Monitor che assicura una ottima gestione di sicurezza e autorizzazione per ogni servizio e utente.

Procedendo con il sign-up si ha la possibilità di selezionare 2 dei 3 servizi disponibili.

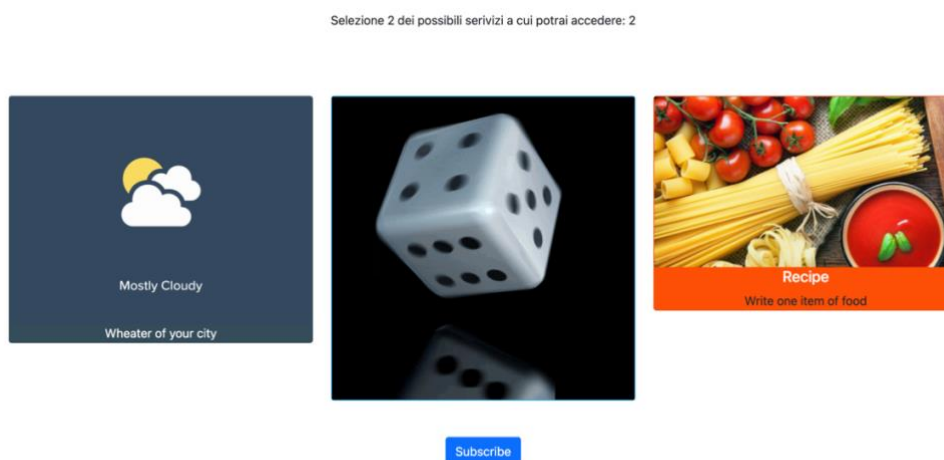


Figura 8 – Servizi disponibili.

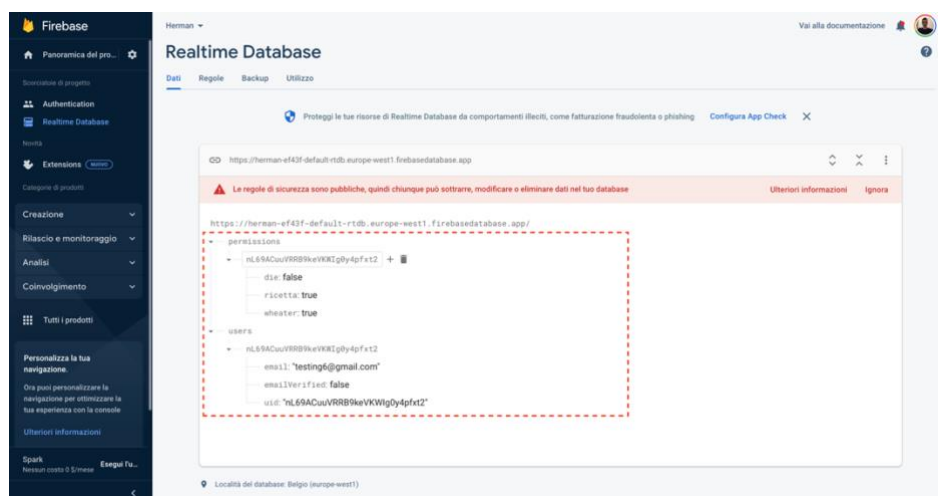


Figura 9 – Real-time Database per la gestione delle autorizzazioni

Questa selezione mostrerà la corretta ed effettiva gestione delle autorizzazioni. Essi vengono salvati su un database (fornito nuovamente da firebase, figura 9), e ad ogni nuova richiesta da parte di un client, il back-end (“ReferenceMonitor” – figura 2) intercetta la richiesta e controlla se l’utente dispone dei permessi adatti per accedere a tale risorsa richiesta: la prima volta effettuerà un accesso al database, successivamente accederà ad una copia locale (cache dei permessi), che verrà generata durante i primi accessi al database da parte degli utenti. Questo è possibile perché i permessi di ogni utente non cambiano mai. Successivamente, in caso di autorizzazione permessa, tramite il protocollo di messaggistica inoltrerà la richiesta ai micro-servizi (“ProtectionObject”, – figura 2), effettuando uno scambio di messaggi. Qui vengono utilizzate funzioni async/await in maniera tale da attendere un risultato, come per esempio il ritorno della risposta da parte del servizio meteo, utilizzando appositi timeout, in maniera tale da non bloccare il sistema. (Promise.race[] [7]).

### 3.1 Sviluppo comunicazione RabbitMQ

In particolare, durante lo sviluppo, sono state implementate due versioni delle possibili implementazioni che offre RabbitMQ:



### 3.1.1 Exchange direct – Filter - Routing

- la prima implementazione (pattern Filter), utilizza l'exchange di tipo *direct*: dopo ogni autorizzazione delle varie richieste da parte dei client, il Reference monitor invia un messaggio fire-and-forget nella coda sulla quale è iscritto il micro-servizio in questione (esempio wheater), successivamente il micro-servizio risponderà su un'altra coda (denominata, per l'esempio precedente: wheater\_reply), dunque il reference monitor rimarrà in attesa della risposta, assumendo il ruolo di consumatore. Qui vengono utilizzate apposite funzioni *async/await* e *Promise*, per gestire il problema del rendering sulla pagina web (front-end).

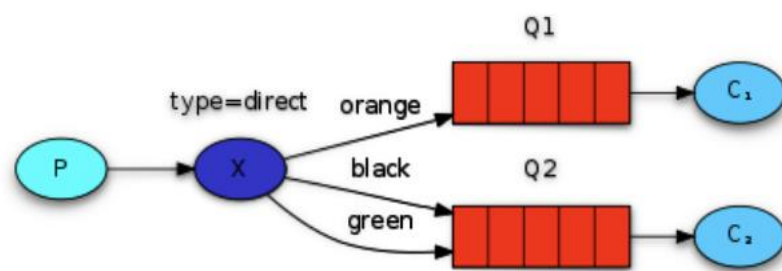


Figura 9 – RabbitMQ – type: direct

### 3.1.2 RPC

- La seconda implementazione (selezionata come soluzione definitiva) utilizza la comunicazione RPC:
  - Quando il Client invia un messaggio, inserisce due proprietà: *response\_to*, che è impostato sulla coda di callback e *correction\_id*, che è impostato su un valore univoco per ogni richiesta, generato da un metodo (*generateUuid()*);
  - La richiesta viene inviata a una coda (es: *rpc\_queue\_micro-serv*).
  - Il server è in attesa di richieste su quella coda. Quando viene intercettata una richiesta, esegue il lavoro e invia un messaggio con il risultato al Client, utilizzando la coda del campo *Reply\_to*.
  - Il client attende i dati nella coda di richiamata. Quando viene visualizzato un messaggio, controlla la proprietà *correction\_id*, per accettarsi che il messaggio di risposta è il suo.

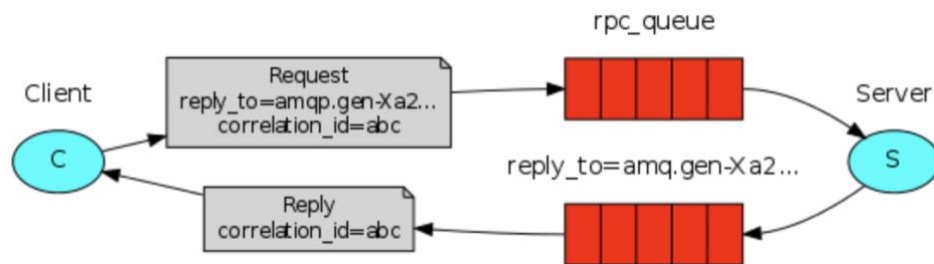


Figura 10 – Modello RPC RabbitMQ

### 3.1.3 Gestione comunicazione con Timeout

Entrambe le implementazioni discusse precedentemente utilizzano il pattern Timeout(). Questo è utile per gestire gli errori che si potrebbero presentare per la non risposta da parte dei micro-servizi, infatti, come mostra la figura 11, la comunicazione con i micro-servizi viene lanciata all'interno del metodo Promise.race() insieme ad una seconda funzione che esegue un timeout di 6000 millisecondi. Nel dettaglio lancia due thread separati: il primo per il metodo che permette la comunicazione con il micro-servizio in questione e il secondo esegue un metodo al cui interno vi è impostato un setTimeout() di 6000 millisecondi. Questo metodo (Promise.race[t1,t2]) ritorna il risultato del thread che finisce per primo l'esecuzione. Dunque se la risposta del servizio eccede i 6 secondi l'utente verrà notificato del mal funzionamento del servizio, garantendo che il sistema non rimarrà in una fase stallo.

```

let setTimeoutConsumer = (mess,item) => {
  return new Promise( resolve => {
    Promise.race([send_rpc_request(mess,item), timeout(6000)]).then((value) => {
      console.log(value);
      resolve(value);
    });
  });
};

```

Figura 11 – Gestione errori con Timeout()

```

app.get('/food', async (req, res) => {
  console.log("Richiesto ricetta per:");
  console.log(req.query.item);
  if(typeof authorization_user[req.query.user]?.["ricetta"] != "undefined") {
    console.log("[*] - Adesso sto controllando l'autorizzazione per ricetta")
    if(authorization_user[req.query.user].ricetta == true) {
      //RABBITMQ: TOPIC
      // send("food", "get_food", req.query.item);
      // let food = await setTimeoutConsumer(['food_result']);

      //RABBITMQ: RPC
      let food = await setTimeoutConsumer( "get_food", req.query.item);
    }
  }
});

```

Figura 12 – esecuzione comunicazione all'interno di Promise.race[]

### 3.2 Diagrammi UML e diagramma di flusso

Di seguito sono disponibili i diagrammi UML e di flusso del seguente software:

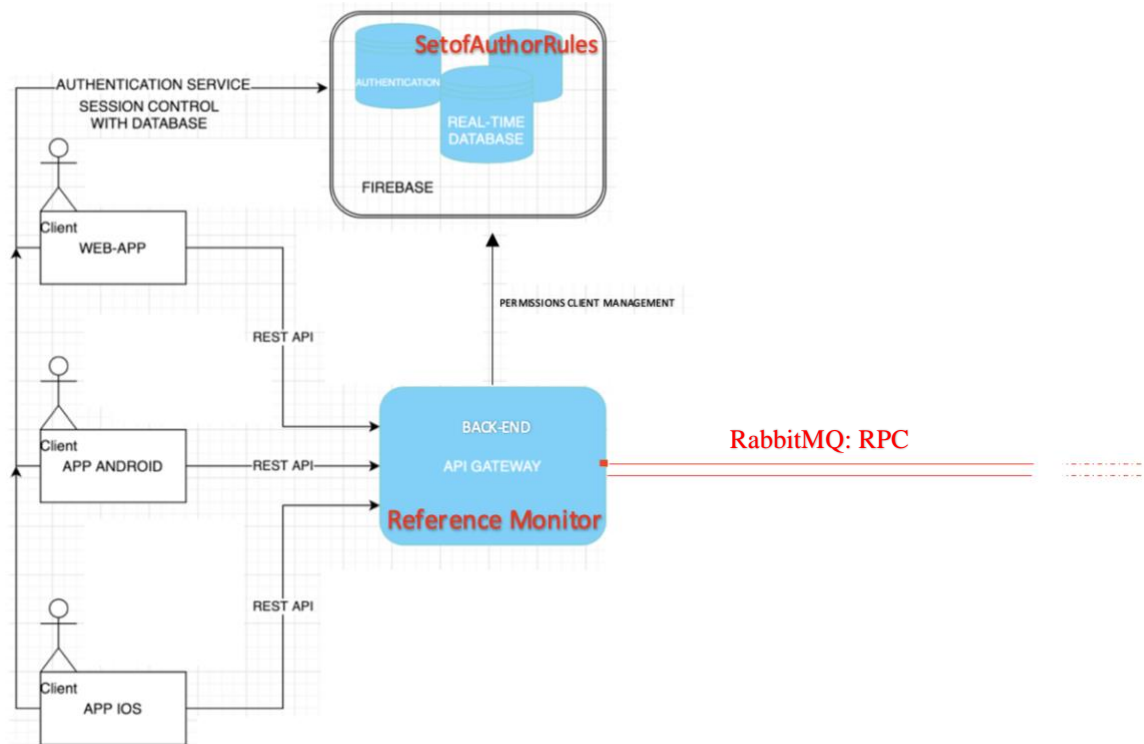


Figure 10 – Diagramma UML (1)

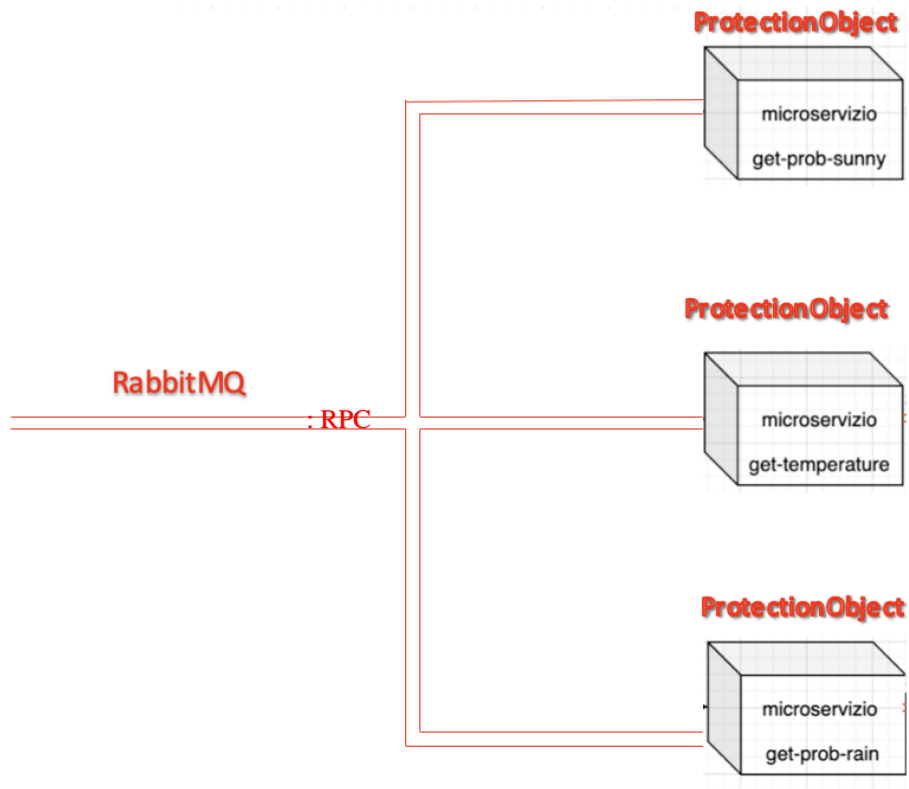


Figure 11 – Diagramma UML (2)

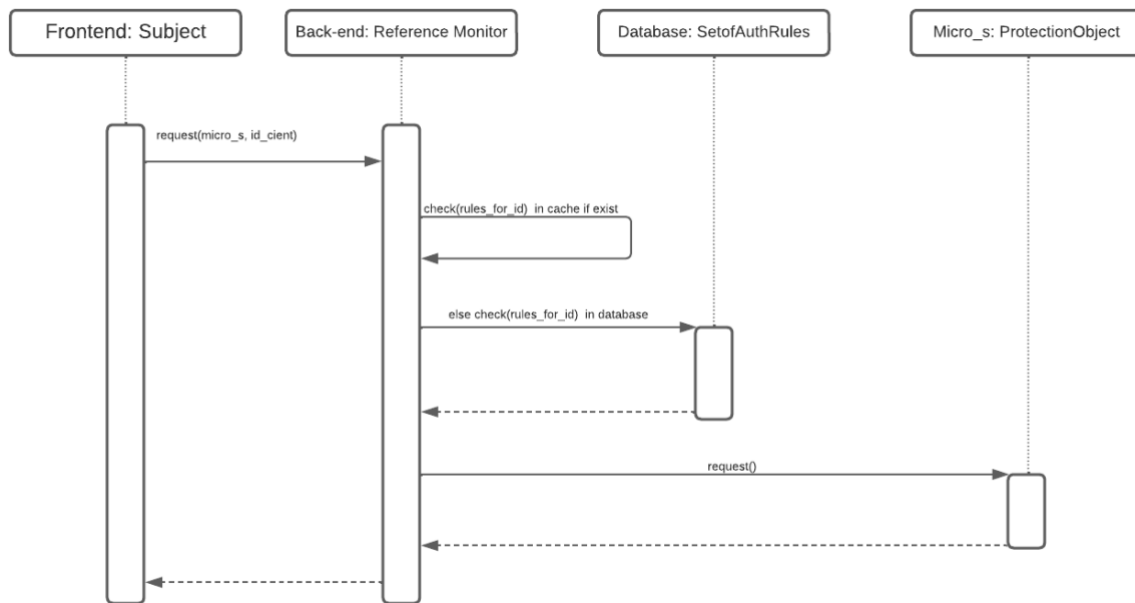


Figura 12 – Diagramma di flusso

### 3.3 Dockerizzazione

Nell'ultima fase del progetto è stata effettuata una dockerizzazione del software, creando 5 diversi container, ciascuno per ogni componente (front-end, back-end, microservice-food, microservice-wheater, microservice-die). In aggiunta sono stati utilizzati altri due container: il primo è il broker RabbitMQ (avviabile anche senza docker tramite il comando "rabbitmq-server" direttamente da terminale), il secondo utile per la rappresentazione visiva dei container sui vari nodi dello swarm (l'immagine in questione è "visualizer", disponibile sulla repository ufficiale docker hub). Le macchine utilizzate sono 2: una macchina con Ubuntu che funge da worker e la macchina Mac da master dello swarm.

Per creare lo swarm bisogna lanciare i seguenti comandi sul terminale:

*docker swarm init*

Facoltativamente, è possibile aggiungere uno o più macchine worker tramite il comando fornito dal nodo master (sopra) dello swarm:

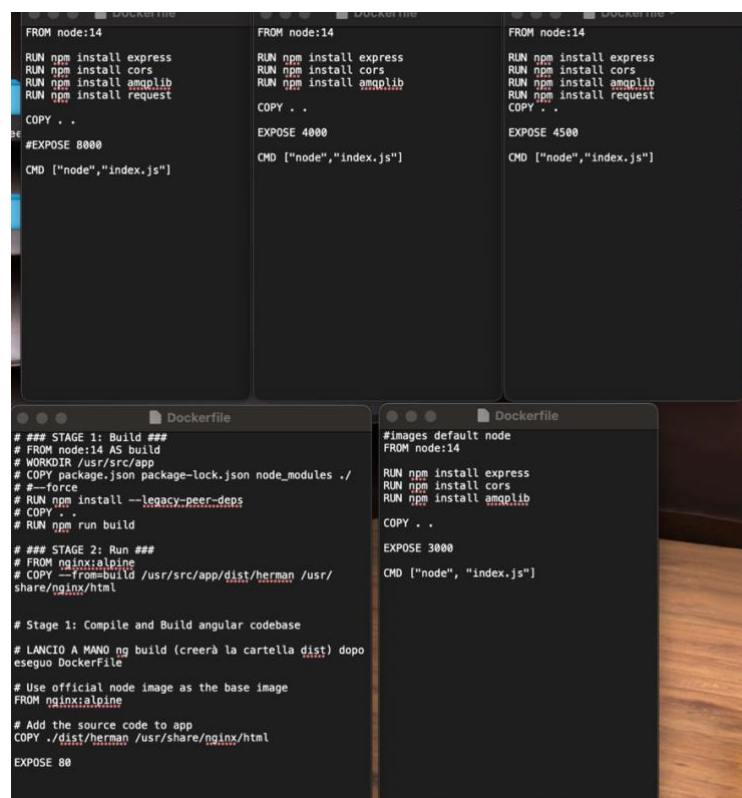
```
docker swarm join --token SWMTKN-1-03dc9ndfptpkaamgdve1riyn07ny4weybelwphe7f106x19xow-abx45z6d4coxf3e622kthm0h192.168.65.3:2377
```

Infine si procede effettuando il deploy del docker-compose.yml sullo swarm:

```
docker stack deploy -c docker-compose.yml projectisd
```

Per accedere al pannello di controllo offerto da Visualizer bisogna recarsi all'indirizzo: <http://localhost:8080>

Di seguito vi sono le immagini dei Dockerfile di ogni componente e del file docker-compose.yml:



```
FROM node:14
RUN npm install express
RUN npm install cors
RUN npm install amqplib
RUN npm install request
COPY . .
#EXPOSE 8000
CMD ["node","index.js"]

FROM node:14
RUN npm install express
RUN npm install cors
RUN npm install amqplib
COPY . .
EXPOSE 4000
CMD ["node","index.js"]

FROM node:14
RUN npm install express
RUN npm install cors
RUN npm install amqplib
RUN npm install request
COPY . .
EXPOSE 4500
CMD ["node","index.js"]

### STAGE 1: Build ###
# FROM node:14 AS build
# WORKDIR /usr/src/app
# COPY package.json package-lock.json node_modules ./
# --force
# RUN npm install --legacy-peer-deps
# COPY . .
# RUN npm run build

### STAGE 2: Run ###
# FROM nginx:alpine
# COPY --from=build /usr/src/app/dist/herman /usr/share/nginx/html

# Stage 1: Compile and Build angular codebase
# LANCIO A MANO ng build (creerà la cartella dist) dopo
# eseguo Dockerfile

# Use official node image as the base image
FROM nginx:alpine

# Add the source code to app
COPY ./dist/herman /usr/share/nginx/html
EXPOSE 80

#images default node
FROM node:14
RUN npm install express
RUN npm install cors
RUN npm install amqplib
COPY . .
EXPOSE 3000
CMD ["node", "index.js"]
```

Figura 14 – DockerFiles.

```

1  version: "3"
2  networks: # definisce, per i container, la rete webnet con eventuali parametri
3    webnet: # qui parametri di default (rete overlay, load-balanced)
4  services:
5    rabbitmq3:
6      container_name: "rabbitmq"
7      image: rabbitmq:3.8-management-alpine
8      environment:
9        - RABBITMQ_DEFAULT_USER=guest
10       - RABBITMQ_DEFAULT_PASS=guest
11      ports:
12        # AMQP protocol port
13        - '5672:5672'
14        # HTTP management UI
15        - '15672:15672'
16    visualizer:
17      image: dockersamples/visualizer:stable # visualizer è disponibile dall'hub Docker
18      ports: # visualizza uno schema dello swarm e,
19        - "8080:8080" # deve girare sul manager (v. sotto deployment)
20      volumes: # la chiave volumes mappa la
21        - "/var/run/docker.sock:/var/run/docker.sock" # socket docker del visualizer
22      deploy:
23        placement:
24          constraints: [node.role == manager] # visualizer deve girare sul manager
25      networks: # visualizzare una rappresentazione
26        - webnet
27    wheater: # al servizio verrà dato il nome dell'immagine con questo suffisso: get-
28      image: giomar19/wheater:latest # immagine eseguita dai container/repliche
29      # user/name:tag individuano l'immagine desiderata
30      deploy: # specifiche per il deployment delle repliche
31        replicas: 1 # n. di repliche
32        resources:
33          limits: # limiti di uso delle risorse che nessuna replica può superare
34            cpus: "0.1" # NB: 0.1=10% della CPU (su tutti i core)
35            memory: 50M
36          restart_policy: # immediately restart any
37            condition: # container, should
38              on-failure # it fail
39        ports:
40          - "4500:4500" # mappa port host:container (- introduce elemento di una
41        networks:
42          - webnet
43  > die: # al servizio verrà dato il nome dell'immagine con questo suffisso: get-sta
44  > food: # al servizio verrà dato il nome dell'immagine con questo suffisso: get-sta
45  > back: # al servizio verrà dato il nome dell'immagine con questo suffisso: get$...
46  > front: # al servizio verrà dato il nome dell'immagine con questo suffisso: get$...

```

Figura 15 – Docker-Compose.yml

## 4. Test e Risultati

È stata effettuata un'accurata fase di testing. Diversi scenari sono stati presi sotto esame:

- testing delle componenti per la gestione delle fasi di signin, signout e session;
- testing sull'utilizzo contemporaneo del software da parte di diversi client;
- testing sulla corretta gestione delle autorizzazioni per ogni utente (corretta applicazione Reference monitor);
- testing sul corretto funzionamento dei singoli container, e dunque dell'intero progetto sul docker-swarm;

Di seguito vi sono alcune immagini che mostrano il corretto funzionamento del software:

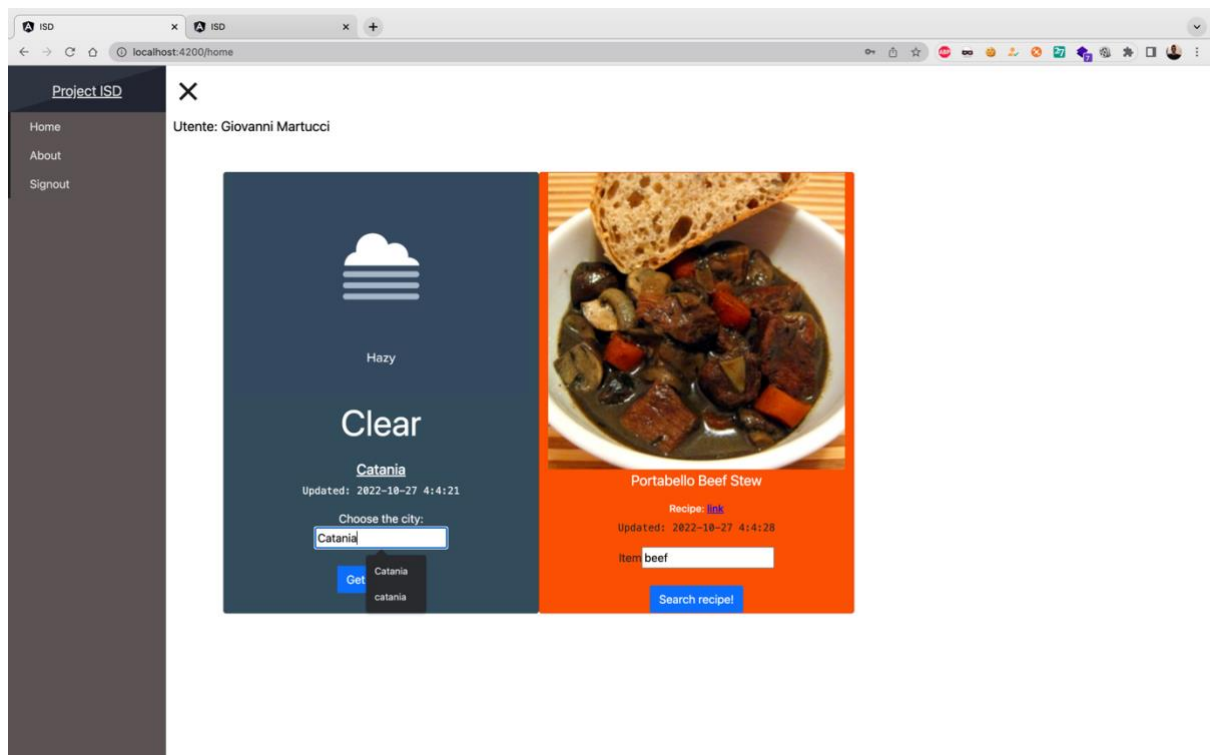


Figura 16 – Home utente 1 (1)



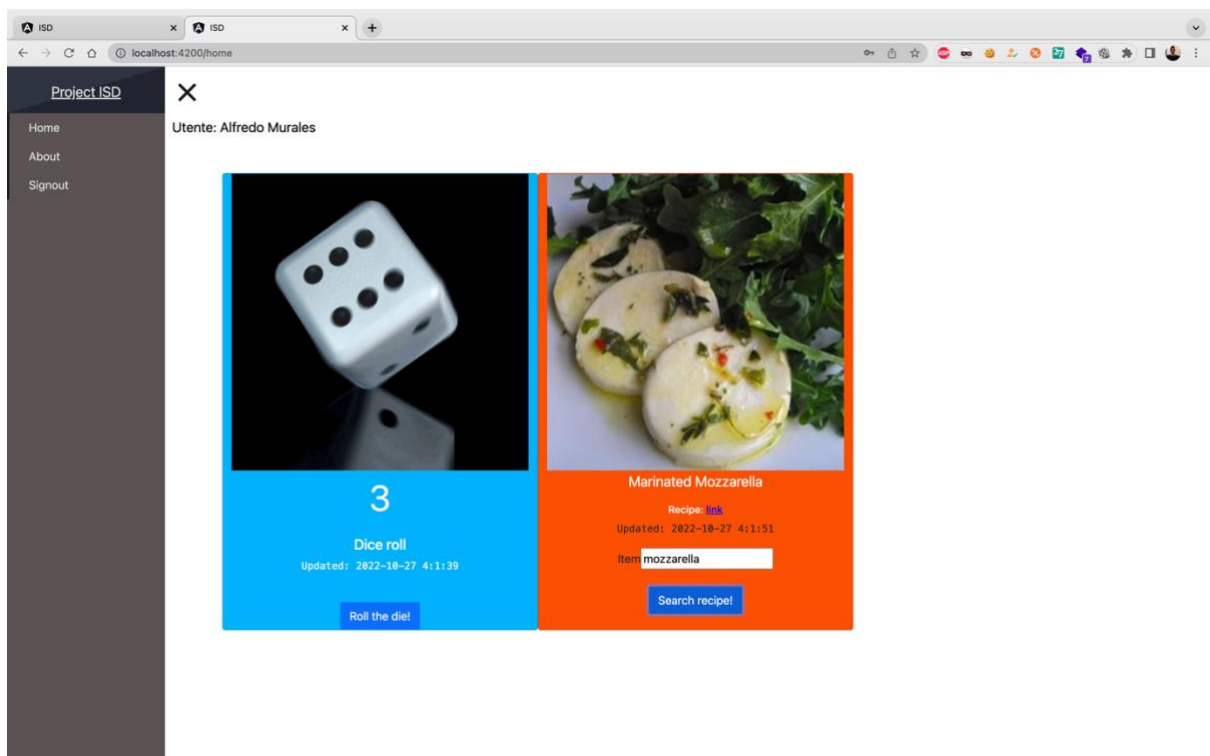


Figura 16 – Home utente 2 (1)

## 5. Usage

- Per l'utilizzo senza Docker bisogna effettuare i seguenti passaggi:

1. Install Node.js;
2. Install Angular;
3. Install RabbitMQ from homebrew;
4. Terminal: rabbitmq-server ;
5. New terminal:
  - cd frontend
  - node index.js ;
6. New terminal:
  - cd backend
  - node index.js ;
7. New terminal:
  - cd microservice\_food\_advice
  - node index.js ;
8. New terminal:
  - cd microservice\_wheater
  - node index.js ;
9. New terminal:
  - cd microservice\_rand\_number
  - node index.js ;
10. Go to <http://localhost:4200> ;

- Utilizzo tramite Docker swarm:

- 1) New terminal: docker swarm init;
- 2) [OPTIONAL] New machines (node worker) terminal: insert code provided by master;
- 3) Terminal master: docker stack deploy -c docker-compose.yml projectisd;
- 4) Go to <http://localhost> ;

\* GO TO <http://localhost:15672> for RabbitMQ panel (user:guest;password:guest);

\* GO TO <http://localhost:8080> for displaying the containers distributed in the various nodes of the swarm;

## Conclusione

Tale progetto ha permesso l'acquisizione di diverse competenze pratiche, analizzate durante il corso di Ingegneria dei sistemi distribuiti. Si è presa consapevolezza dei principali design pattern utilizzati al giorno d'oggi, in particolare, in questo progetto, è stato implementato il pattern Reference Monitor. Tra le metodologie di messaggistica presenti, è stata utilizzato RabbitMQ. Si conclude affermando che, avere delle ottime conoscenze dei design pattern e delle best-practice esistenti per lo sviluppo di applicazioni software, garantisce un'eccellente implementazione e progettazione dell'intero software.

## Reference

- [1] RabbitMQ, "RabbitMQ," 7 Ottobre 2022. [Online]. Available: <https://www.rabbitmq.com/>.
- [2] E. F.-B. D. H. F. B. P. S. Markus Schumacher, Secutiy Patterns, Wiley, 2006.
- [3] Docker, "Docker," [Online]. Available: <https://www.docker.com/>. [Accessed 9 October 2022].
- [4] Angular, "Angular," [Online]. Available: <https://angular.io/>. [Accessed 6 October 2002].
- [5] "Nodejs," [Online]. Available: <https://nodejs.org/en/>.
- [6] Google, "firebase," [Online]. Available: <https://firebase.google.com/>.
- [7] mozilla, "developer mozilla," [Online]. Available: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise/race?retiredLocale=it](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/race?retiredLocale=it). [Accessed October 2002].

## Repository GitHub

Link disponibile: