# Assignment #4

Martina Magnani
martina.magnani8@studio.unibo.it

Nicola Piscaglia
nicola.piscaglia2@studio.unibo.it

Mattia Vandi
mattia.vandi@studio.unibo.it

## 1 Analisi del problema

Implementare un servizio di chat distribuito, prendendo come riferimento le architetture orientate ai servizi e reattive e aspetti metodologici correlati.
Il servizio di chat deve permettere di creare dinamicamente nuove chat, per ognuna delle quali valgono i requisiti specificati nell'Assignment 03:

- Un utente può aggiungersi e rimuoversi dinamicamente.
- Ogni messaggio inviato da un utente deve essere visualizzato da tutti gli altri utenti.
- Il sistema deve essere completamente decentralizzato.
  - A parte, eventualmente, la presenza di un (attore) registro con indirizzo/nome noto che tenga traccia dei partecipanti.
- I messaggi inviati nella chat devono essere visualizzati da tutti i partecipanti nel medesimo ordine.

Supporto per una modalità "sezione critica":

- Un partecipante può chiedere di entrare in sezione critica inserendo un comando predefinito.
- Quando un partecipante entra in sezione critica, possono essere visualizzati solo i suoi messaggi, senza intervallarli a quelli degli altri utenti.
- Un solo partecipante alla volta può essere in sezione critica.
- Per uscire dalla sezione critica si può prevedere un altro comando predefinito (es: ":exit-cs").
- Un utente può rimanere in sezione critica per un certo tempo massimo Tmax, dopodiché l'uscita è forzata.
- Durante una sezione critica i messaggi inviati dagli altri partecipanti devono essere rigettati.

Il servizio può avere un front-end di qualsiasi tipo (a linea di comando, GUI, web-based).
Il servizio deve essere progettato mettendo in campo un'architettura che consideri aspetti non funzionali quali:

- scalabilità (all'aumentare del numero di chat e del numero di utenti per chat),
- resilienza,
- responsività.

# 2 Descrizione della soluzione proposta

## 2.1 Architettura del sistema

Ad alto livello il sistema è formato da due macrocomponenti: un *Web Server* ed un *Web Client* (di seguito chiamato `ChatClient`). Il *Web Server* è stato organizzato utilizzando l'architettura a microservizi. In particolare, i servizi presenti nella nostra soluzione sono:

- `UserService`: gestisce le operazioni di memorizzazione/lettura dei dati degli utenti.
- `RoomService`: gestisce lo stato delle chat distribuite.
- `WebAppService`: gestisce la comunicazione del client web con gli altri servizi.
- `HealthCheckService`: controlla la disponibilità dei vari servizi.

## 2.2 Dinamica del sistema

Ogni servizio espone una propria interfaccia REST per poter essere contattato dagli altri servizi. La dinamica del sistema è stata rappresentata ad alto livello con diagrammi di sequenza, adatti per mostrare in modo efficace lo scambio di messaggi tra le entità del sistema.
Ogni interazione fra servizi è preceduta da una verifica della disponibilità del servizio effettuata dall'`HealthCheckService`.
Di seguito vengono mostrati i principali scenari d'interazione.

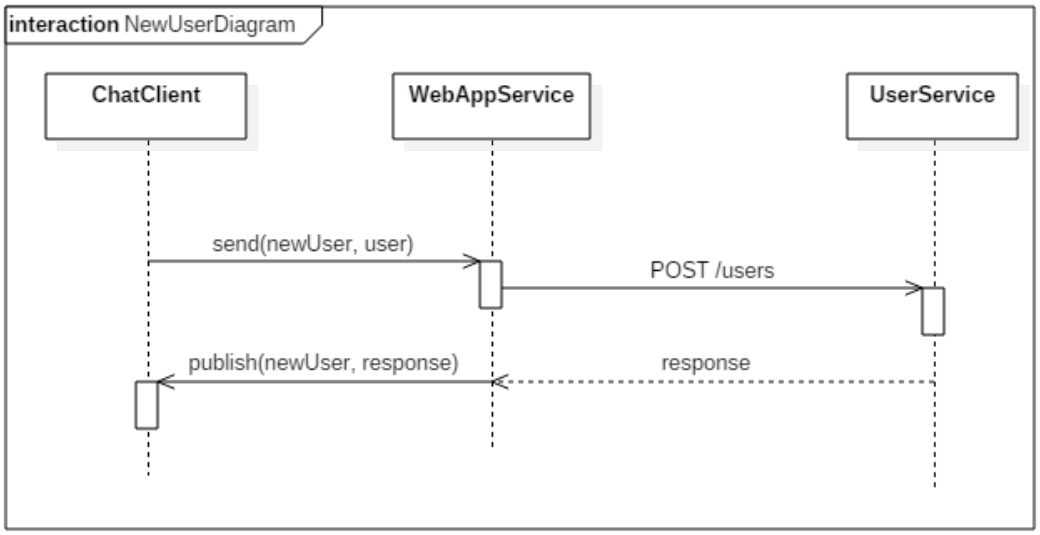


Figura 1: Diagramma di sequenza che mostra la registrazione di un nuovo utente.

Nell'interazione descritta in figura 1 l'interazione fra i vari servizi è la seguente: (1) l'utente effettua il login sull'applicazione client (`ChatClient`); (2) il client pubblica l'evento sull'Event Bus; (3) il servizio (`WebAppService`) in ascolto sul canale inoltra il messaggio allo `UserService`; (4) lo `UserService` crea il nuovo utente e risponde con il risultato dell'operazione.

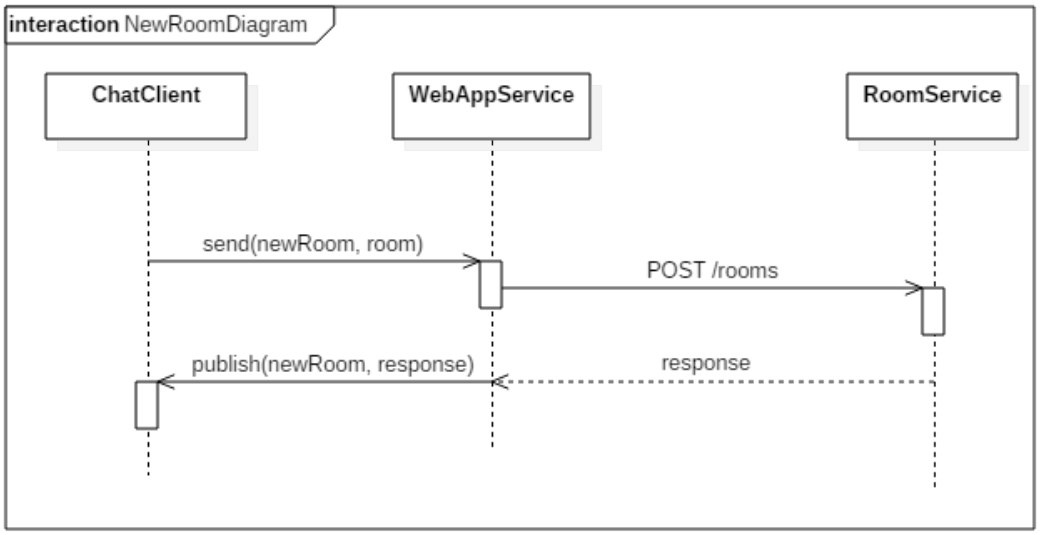


Figura 2: Diagramma di sequenza che mostra la creazione di una stanza.

Nell'interazione descritta in figura 2 la sequenza di messaggi scambiati è la sequente:

(1) il client pubblica l'evento sull'Event Bus;

(2) il servizio (`WebAppService`) in ascolto sul canale inoltra il messaggio al `RoomService`;

(3) il `RoomService` crea la nuova stanza e risponde con il risultato dell'operazione.

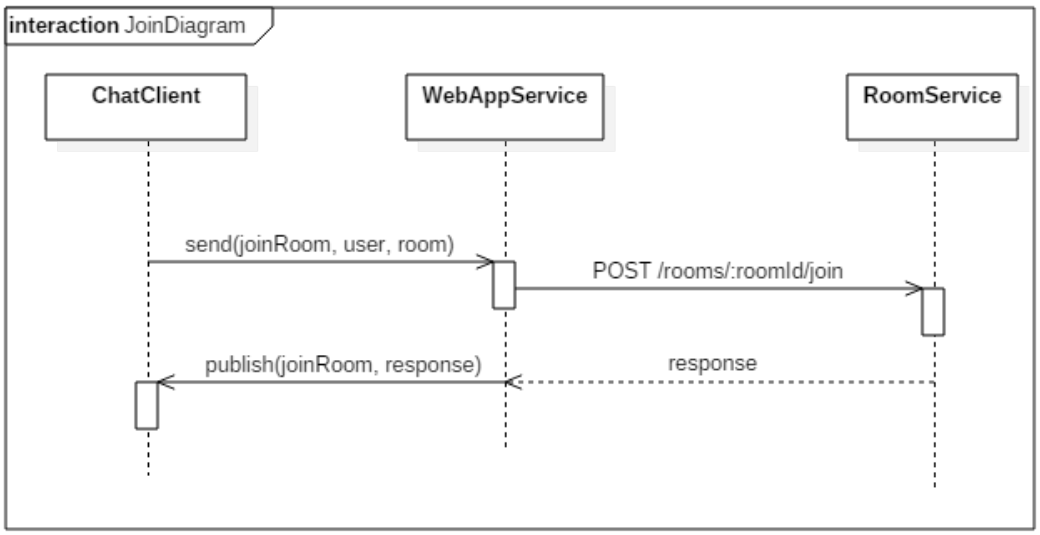


Figura 3: Diagramma di sequenza che mostra l'entrata di un utente in una stanza.

Nell'interazione descritta in figura 3 la sequenza di messaggi scambiati è la sequente: (1) il client pubblica l'evento sul canale, mostrando l'intenzione dell'utente di entrare/uscire dalla stanza; (2) il `WebAppService` in ascolto sul canale inoltra il messaggio al `RoomService`; (3) il `RoomService` aggiunge/rimuove l'utente alla/dalla stanza e risponde con il risultato dell'operazione.

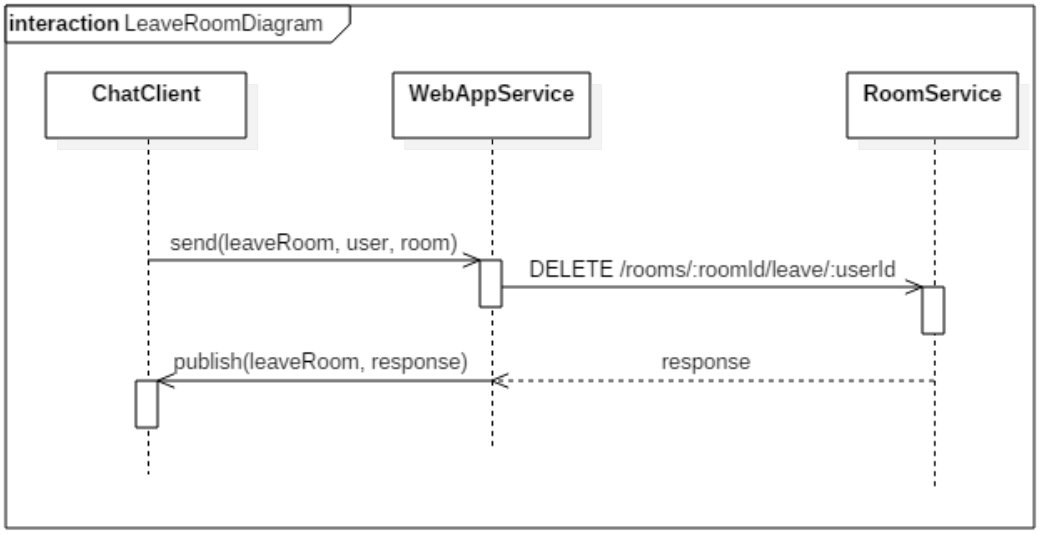


Figura 4: Diagramma di sequenza che mostra l'uscita di un utente da una stanza.

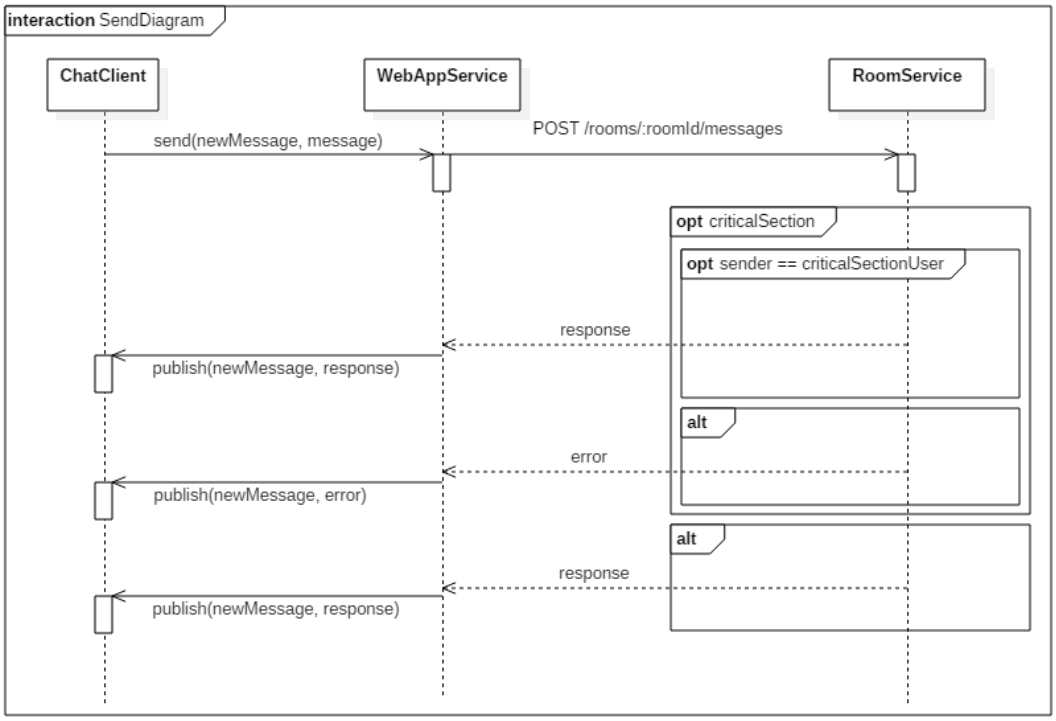


Figura 5: Diagramma di sequenza che mostra lo scambio di messaggi fra gli utenti di una stanza.

Nell'interazione descritta in figura 5 la sequenza di messaggi scambiati è la sequente:

(1) Il client pubblica l'evento sull'Event Bus, mostrando l'intenzione di un utente di pubblicare un messaggio nella chat.

(2) il `WebAppService` in ascolto sul canale inoltra il messaggio al `RoomService`;

(3) Nel caso in cui non ci sia una sezione critica il messaggio viene inviato a tutti gli altri client.

(4) Nel caso in cui ci sia una sezione critica:

   (a) se il mittente è l'utente in sezione critica il messaggio viene accettato;

   (b) se il mittente non è l'utente in sezione critica, il messaggio viene rifiutato e viene segnalata l'impossibilità di mandare il messaggio.

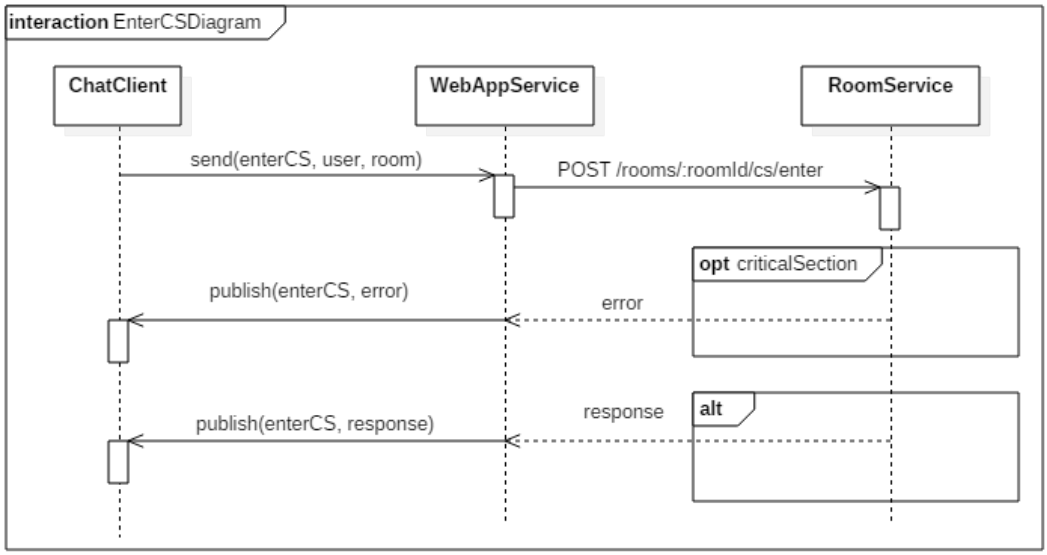


Figura 6: Diagramma di sequenza che mostra la richiesta di entrare in sezione critica da parte di un utente.

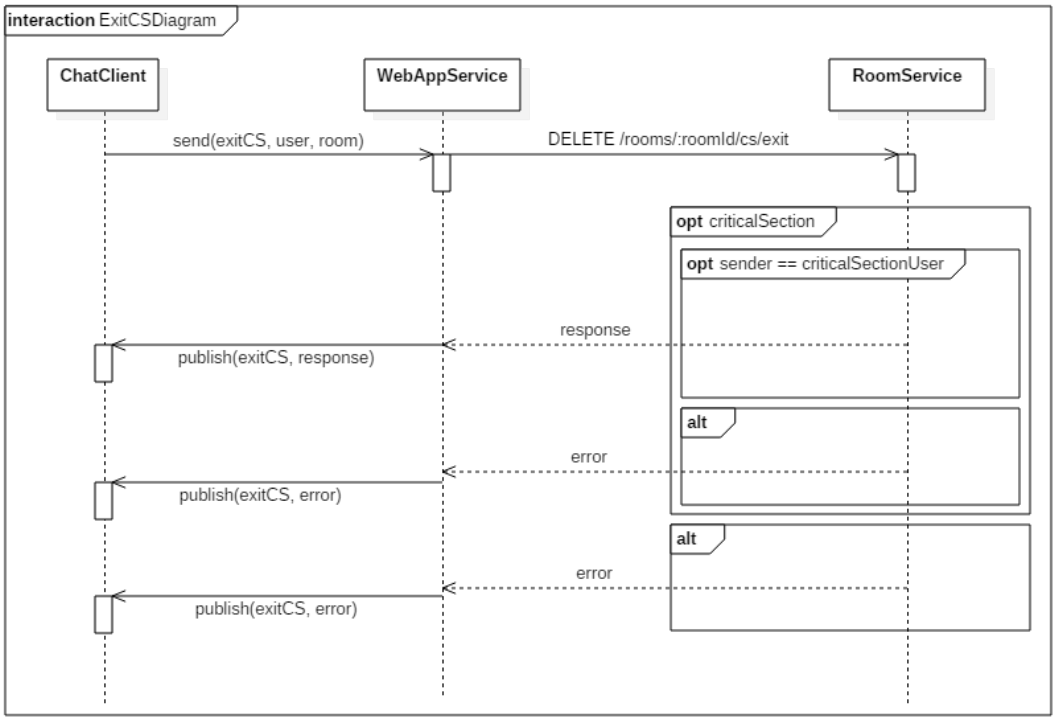


Figura 7: Diagramma di sequenza che mostra la richiesta di uscire dalla sezione critica da parte di un utente.

## 2.3 Algoritmo di ordinamento

Il problema dell'ordinamento dei messaggi può essere scomposto in due sottoproblemi tra loro indipendenti:

- **causal message ordering**: assicura che tutti i messaggi vengano elaborati secondo il loro ordine di invio;
- **total message ordering**: assicura che tutti i processi elaborino i messaggi nello stesso ordine.

Il problema dell'ordinamento dei messaggi è stato risolto utilizzando due algoritmi che descriviamo di seguito. Questo permette ad ogni utente di avere una visione coerente e ordinata dei messaggi.

## 2.4 Causal Message Ordering

Ogni messaggio è costituito dal testo e dal numero di sequenza associatogli dal mittente. Ogni utente mantiene per ogni processo il numero di messaggi inviati e viene aggiornato ogni volta che l'utente è abilitato a processare il messaggio ricevuto.
Un utente è abilitato ad elaborare il messaggio quando il numero di sequenza all'interno del messaggio è successivo al numero di sequenza dell'ultimo messaggio elaborato per quel mittente.

Se così non fosse, significa che ci sono dei messaggi precedenti del mittente che non sono ancora stati ricevuti ed è quindi necessario attendere il loro arrivo prima di potere elaborare il messaggio appena ricevuto.

1. Un processo che si unisce al sistema riceve da tutti gli altri processi il loro numero di sequenza corrente.
2. Un processo che invia un messaggio incrementa il proprio numero di sequenza e lo associa al messaggio.
3. Un processo che riceve un messaggio controlla che il numero di sequenza contenuto nel messaggio sia successivo all'ultimo numero di sequenza che ha ricevuto dal mittente.
   Se tale condizione è soddisfatta:
   (a) il messaggio viene elaborato
   (b) se ci sono altri messaggi che possono essere elaborati si ritorna al punto precedente.

   Se tale condizione non è soddisfatta il messaggio viene aggiunto alla coda dei messaggi in attesa.
4. Un processo che esce dal sistema viene rimosso dalle liste di tutti gli altri processi.

## 2.5 Total Message Ordering

I processi inviano i messaggi al *Sequencer*: entità che assegna un numero di sequenza globale ad ogni messaggio. Ogni processo mantiene un contatore che rappresenta il numero di sequenza globale dell'ultimo messaggio elaborato.
Quando un utente invia un messaggio:

1. Il *Sequencer* assegna al messaggio un numero di sequenza.
2. Un processo che riceve un messaggio controlla che il numero di sequenza del messaggio sia successivo all'ultimo numero di sequenza salvato.
   Se tale condizione è soddisfatta:
   (a) il messaggio viene elaborato
   (b) se ci sono altri messaggi che possono essere elaborati si ritorna al punto precedente.

   Se tale condizione non è soddisfatta il messaggio viene aggiunto alla coda dei messaggi in attesa.

## 2.6 Implementazione

La soluzione è stata implementata in linguaggio Java utilizzando la libreria Vert.x. È stato scelto di utilizzare Vert.x perché permette di implementare i microservizi e i pattern per la resilienza e la scalabilità in modo agile e conciso. Inoltre, rispetto a Spring e ad altri web-framework, Vert.x è una libreria leggera, asincrona e performante.
I servizi presenti nel sistema sono gli utenti della chat (gestiti dallo `UserService`), il gestore delle varie chat (rappresentato dal `RoomService`) e il servizio che si occupa dell'interazione con il client (rappresentato dal `WebAppService`).

### 2.6.1 Interfacce di comunicazione

Ogni servizio espone un'interfaccia REST HTTP che lo rende contattabile dall'esterno. Certe chiamate richiedono la trasmissione di oggetti JSON e in alcuni casi prevedono dei parametri nell'URI (preceduti dai due punti ':'). Di seguito mostriamo i metodi esposti da ogni servizio.

#### 2.6.1.1 UserService

- `POST /users`: metodo che permette la creazione di un nuovo utente.
  - Richiede in ingresso un JSON contenente il nome dell'utente da creare.

```
{
    "name": "userName"
}
```

  - Restituisce un JSON contenente il nome e l'identificativo del nuovo utente creato.

```
{
    "name": "userName",
    "id": 1
}
```

- `GET /users`: metodo che restituisce tutti gli utenti presenti nel sistema.
  - Restituisce un array di JSON contenente gli utenti.

```
[
    {
        "name": "userName",
        "id": 1
    },

    {
        "name": "userName",
        "id": 2
    },

    ...
]
```

- `GET /users/:id`: metodo che restituisce un utente dato il suo identitificativo (id).
  - Restituisce il JSON contenente l'utente richiesto.

```
{
    "name": "userName",
    "id": 1
}
```

- `PUT /users/:id`: metodo che modifica le informazioni di un utente dato il suo identitificativo (id)
  - Richiede in ingresso un JSON contente il nuovo nome dell'utente da modificare.

    ```
    {
        "name": "newUserName"
    }
    ```

  - Restituisce un JSON contenente l'utente modificato.

    ```
    {
        "name": "newUserName",
        "id": 1
    }
    ```

- `DELETE /users/:id`: metodo che elimina un utente dato il suo identificativo (id).
  - Restituisce un JSON contenente l'identificativo dell'utente eliminato.

    ```
    {
        "id": 1
    }
    ```

#### 2.6.1.2 RoomService

- `POST /rooms`: metodo che permette la creazione di una nuova stanza.
  - Richiede in ingresso un JSON contenente il nome della stanza da creare.

    ```
    {
        "name": "roomName"
    }
    ```

  - Restituisce un JSON contenente il nome e l'identificativo della nuova stanza creata.

    ```
    {
        "name": "roomName",
        "id": 1
    }
    ```

- `GET /rooms`: metodo che restituisce tutte le stanze presenti nel sistema.
  - Restituisce un JSONArray contenente le stanze.

    ```
    [
        {
            "name": "roomName",
            "id": 1
        },
    ```

```
    {
        "name": "roomName",
        "id": 2
    },

    ...
]
```

- `GET /rooms/:id`: metodo che restituisce una stanza dato il suo identitificativo (id).
  - Restituisce il JSON contenente la stanza richiesta.

```
{
    "name": "roomName",
    "id": 1
}
```

- `PUT /rooms/:id`: metodo che modifica le informazioni di una stanza dato il suo identitificativo (id).
  - Richiede in ingresso un JSON contente il nuovo nome della stanza da modificare.

```
{
    "name": "newRoomName"
}
```

  - Restituisce un JSON contenente la stanza modificata.

```
{
    "name": "newRoomName",
    "id": 1
}
```

- `DELETE /rooms/:id`: metodo che elimina una stanza dato il suo identificativo (id).
  - Restituisce un JSON contenente l'identificativo della stanza eliminata.

```
{
    "id": 1
}
```

- `POST /rooms/:roomId/messages`: metodo che permette di aggiungere un messaggio alla chat corrispondente alla stanza con identificativo `roomId`.
  - Richiede in ingresso un JSON contente l'utente che vuole inviare il messaggio, il suo clock e il contenuto del messaggio.

```
{
    "user": {
        "id": 1,
        "name": "userName"
    },
    "userClock": 3,
    "content": "This is the message content"
}
```

- Restituisce un oggetto JSON uguale a quello ricevuto con in aggiunta il globalCounter.

```
{
    "user": {
        "id": 1,
        "name": "userName"
    },
    "userClock": 3,
    "content": "This is the message content",
    "globalCounter": 5
}
```

- `POST /rooms/:roomId/join`: metodo che permette ad un utente di unirsi ad una stanza.
  - Richiede in ingresso un JSON contente l'utente che vuole unirsi alla stanza.

```
{
    "id": 1,
    "name": "userName"
}
```

  - Restituisce un oggetto JSON contentente il global counter della stanza e la lista di tutti i clock degli utenti all'interno della stanza.

```
{
    "globalCounter": 3,
    "usersClock": [
        {
            "user": user1,
            "userClock": 1
        },

        {
            "user": user2,
            "userClock": 2
        },

        ...
    ]
}
```

- `DELETE /rooms/:roomId/leave/:userId`: metodo che permette ad un utente di abbandonare la stanza a cui si è già unito.
  - Restituisce un oggetto JSON vuoto in caso di successo o un JSON contenente un messaggio di errore in caso di fallimento.
- `GET /rooms/:roomId/cs`: metodo che indica se la stanza è in sezione critica o meno e, in tal caso, qual'è l'utente in sezione critica.
  - Restituisce un oggetto JSON che indica se la sezione critica è stata acquisita o meno e, in tal caso, da quale utente.

    ```
    {
        "held": true,
        "user": {
            "id": 1,
            "name": "userName"
        }
    }
    ```

  - Restituisce un oggetto JSON vuoto in caso di successo o un JSON contenente un messaggio di errore in caso di fallimento.
- `POST /rooms/:roomId/cs/enter`: metodo che permette ad un utente cha ha già acquisito la sezione critica in una stanza di rilasciarla.
  - Richiede in ingresso un JSON contente l'utente che vuole entrare in sezione critica.

    ```
    {
        "id": 1,
        "name": "userName"
    }
    ```

  - Restituisce un oggetto JSON vuoto in caso di successo o un JSON contenente un messaggio di errore in caso di fallimento.
- `DELETE /rooms/:roomId/cs/exit/:userId`: metodo che permette l'uscita dalla sezione critica.
  - Restituisce un oggetto JSON vuoto in caso di successo o un JSON contenente un messaggio di errore in caso di fallimento.

#### 2.6.1.3 WebAppService

- `POST /messages`: metodo che permette di notificare il `WebAppService` qualora il timeout della sezione critica sia scaduto.
  - Richiede un oggetto JSON contenente la stanza in cui è scaduta la sezione critica.

    ```
    {
        "id": 1,
        "name": "userName"
    }
    ```

### 2.6.2 WebAppService e ChatClient

Il `WebAppService` è il servizio che permette la comunicazione con il client (`ChatClient`). La comunicazione avviene attraverso l'Event Bus di Vert.x. Abbiamo definito un unico canale di comunicazione dal client verso il servizio e più canali di comunicazione dal servizio verso il client (uno per ogni risposta).
Ogni messaggio che il client pubblica sull'Event Bus è definito da un JSON contenente il *tipo* del messaggio e la *richiesta*. Di seguito descriviamo tutti i tipi di messaggi inviati dal client e le relative risposte. Vengono inoltre mostrati esempi delle richieste e risposte contenute nei messaggi.

- `newUser`: messaggio pubblicato al momento del login di un nuovo utente.
  - Contenuto del messaggio di invio:

```
{
    "type": "newUser",
    "request": {
        name: "userName"
    }
}
```

  - Contenuto del messaggio di risposta:

```
{
    "id": 1,
    "name": "userName"
}
```

- `deleteUser`: messaggio pubblicato al momento del logout di un utente.
  - Contenuto del messaggio di invio:

```
{
    "type": "deleteUser",
    "request": {
        "userId": 1
    }
}
```

  - Contenuto del messaggio di risposta:

```
{
    "userId": 1
}
```

- `rooms` : messaggio pubblicato quando un utente effettua il login in modo da ottenere le eventuali stanze presenti:
  - Contenuto del messaggio di invio:

```
{
    "type": "rooms"
}
```

– Contenuto del messaggio di risposta:

```
[
    {
        "id": 1,
        "name": "roomName"
    },

    {
        "id": 2,
        "name": "roomName"
    },

    ...
]
```

- `newRoom`: messaggio pubblicato quando viene aggiunta una nuova stanza.
  – Contenuto del messaggio di invio:

```
{
    "type": "newRoom",
    "request": {
        "name": "roomName"
    }
}
```

– Contenuto del messaggio di risposta:

```
{
    "id": 1,
    "name": "roomName"
}
```

- `deleteRoom`: messaggio pubblicato quando un utente elimina una stanza.
  – Contenuto del messaggio di invio:

```
{
    "type": "deleteRoom",
    "request": {
        "roomId": 1
    }
}
```

- Contenuto del messaggio di risposta:

```
{
    "roomId": 1
}
```

- `joinRoom`: messaggio pubblicato quando un utente entra in una stanza:
  - Contenuto del messaggio di invio:

```
{
    "type": "joinRoom",
    "request": {
        "roomId": 1,
        "user": user
    }
}
```

  - Contenuto del messaggio di risposta:

```
{
    "roomId": 1,
    "user": user,
    "globalCounter": 10,
    "usersClock": [
        {
            "userId": 1,
            "userClock": 0
        },
        {
            "userId": 2,
            "userClock": 4
        },
        ...
    ]
}
```

- `leaveRoom`: messaggio pubblicato quando un utente esce dalla stanza.
  - Contenuto del messaggio di invio:

```
{
    "type": "leaveRoom",
    "request": {
        "roomId": 1,
        "user": user
    }
}
```

  - Contenuto del messaggio di risposta:

```
{
    "type": "leaveRoom",
    "request": {
        "roomId": 1,
        "user": user
    }
}
```

- `newMessage`: messaggio pubblicato quando un utente invia un messaggio nella chat:
  - Contenuto del messaggio di invio:

```
{
    "type": "newMessage",
    "request": {
        "roomId": 1,
        "user": user,
        "userClock": 0,
        "content": "Message content"
    }
}
```

  - Contenuto del messaggio di risposta:

```
{
    "roomId": 1,
    "user": user,
    "userClock": 0,
    "content": "Message content",
    "globalCounter": 3
}
```

- `enterCS`: messaggio pubblicato quando un utente invia il comando che permette di entrare in sezione critica:
  - Contenuto del messaggio di invio:

```
{
    "type": "enterCS",
    "request": {
        "roomId": 1,
        "user": user
    }
}
```

  - Contenuto del messaggio di risposta:

```
{
    "roomId": 1,
    "user": user
}
```

- `exitCS`: messaggio pubblicato quando un utente invia il comando che permette di uscire dalla sezione critica:
  - Contenuto del messaggio di invio:

    ```
    {
        "type": "exitCS",
        "request": {
            "roomId": 1,
            "user": user
        }
    }
    ```

  - Contenuto del messaggio di risposta:

    ```
    {
        "roomId": 1,
        "user": user
    }
    ```

### 2.6.3 ChatClient

Il front-end del sistema è un *Web Client* realizzato con il framework *Angular* e servito dal `WebAppService`. L'app si presenta con una prima pagina di Login, dove l'utente potrà inserire il suo nickname. Una volta effettuato l'accesso la pagina presenta: (1) un menù laterale in cui sono visibili le stanze presenti nel sistema, un bottone che permette la creazione di nuove stanze e un bottone per il logout. (2) la schermata di messaggistica; attiva solo se l'utente si è unito ad una stanza.

## 2.7 Aspetti non funzionali

Di seguito mostriamo le soluzioni adottate per adempiere ai requisiti non funzionali specificati nell'analisi del problema.

### 2.7.1 Responsività

Vert.x implementando i principi descritti dal Reactive Manifesto - ovvero *elasticità*, *resilienza*, *responsività* e *comunicazione orientata ai messaggi* - permette di soddisfare il requisito di responsività.
Vert.x implementa il Pattern *Multi-Reactor*, il quale mantiene più event-loop al fine di permettere ai singoli processi di scalare attraverso i CPU core della macchina sulla quale sta eseguendo l'applicazione. E' possibile specificare il numero di event-loop, di default sono il doppio dei core della CPU della macchina.

### 2.7.2 Resilienza

Per adempiere al requisito di resilienza abbiamo implementato due meccanismi:
(1) `HealthCheckService` per monitorare lo stato dei servizi; e (2) `ServiceManager` per gestire proattivamente il deployment di un servizio e per controllare periodicamente il suo stato, al fine di riesumarlo qualora non fosse più attivo.

- **HealthCheckService**: fornisce un'interfaccia REST che espone i metodi necessari per recuperare informazioni sullo stato di salute di uno specifico servizio e di tutti i servizi da cui dipende. Di seguito le rotte utilizzate per ogni servizio:
  - `GET /health/room`: controlla che il servizio `RoomService` e il servizio da cui dipende (`WebAppService`) siano attivi. Nel caso in cui uno qualsiasi dei due servizi non sia disponibile la risposta sarà: `Error 403 (Service Unavailable)`, fornendo anche il dettaglio sullo stato dei servizi. Nel caso in cui tutti i servizi siano disponibili la risposta sarà: `200 OK`.
  - `GET /health/user`: controlla che il servizio `UserService` sia disponibile. Nel caso in cui non sia disponibile la risposta sarà: `403 Service Unavailable`. Nel caso in cui il servizio sia disponibile la risposta sarà: `200 OK`.
  - `GET /health/webapp`: controlla che il servizio `WebApppService` e i servizi da cui dipende (`RoomService` e `UserService`) siano attivi. Nel caso in cui uno qualsiasi dei tre servizi non sia disponibile la risposta sarà: `403 Service Unavailable`, fornendo anche il dettaglio sullo stato dei servizi. Nel caso in cui tutti i servizi siano disponibili la risposta sarà: `200 OK`.

  Ogni servizio espone una rotta `GET /health` che viene contatta dall'`HealthCheckService` al momento di verifica dello stato dello specifico servizio.
- **ServiceManager**: progettato per monitorare proattivamente un certo servizio. È presente un manager per ogni servizio, il quale effettua periodicamente la chiamata `GET /health` del servizio monitorato. Nel caso il servizio non sia disponibile, il `ServiceManager` si preoccupa di riesumarlo avviando un nuovo processo di sistema corrispondente all'applicazione del servizio.

### 2.7.3 Scalabilità

Ogni servizio gestisce le richieste e ha a disposizione un insieme di `Worker` che hanno il compito di elaborare le richieste effettuate al servizio e di inoltrargli il risultato, che a sua volta verrà comunicato al richiedente. L'interazione tra il servizio e i worker è mediata dall'Event Bus di Vert.x.

# A Comandi

I comandi inviabili ad ogni stanza sono:

- *:enter-cs*: per richiedere l'entrata in sezione critica da parte di un utente
- *:exit-cs*: per richiedere l'uscita dalla sezione critica da parte di un utente
- *:help*: per richiedere la lista dei comandi disponibili