

UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato Web and Real Time Communication

UNINA VIDEO CONFERENCE

Anno Accademico 2021/2022

Alessandro D'Angelo M63001181

Domenico Iorio M63001251

Michelle Pepe M63001196

SOMMARIO

Unina Video Conference	0
1 <i>Introduzione</i>	1
1.1 Node.JS	1
1.2 Janus	2
1.3 Docker	4
2 <i>Architettura</i>	6
2.1 Server	9
2.2 Janus-Gateway	10
2.3 Database	11
3 <i>Funzionamento</i>	12
3.1 Iscrizione e login.....	12
3.2 Start	14
3.3 VIDEOROOM	15
3.4 CASI D'USO	16
4 <i>Janus</i>.....	17
4.1 Connessione al Plugin (Publisher)	18
4.2 Create and Join Room	19
4.3 Gestione Risposte del server	21
4.4 PublishOwnFeed.....	22
4.5 NewRemoteFeed	23
5 <i>Sicurezza</i>	24
5.1 Access Control	24
5.2 Environment Variables	26
5.3 Filtraggio e Validazione.....	26
5.4 Email Confirmation	27

1 INTRODUZIONE

Per l'implementazione e lo sviluppo del progetto “*Unina Video Conference*” si è scelto di utilizzare le seguenti tecnologie:

- Node.JS
- Janus
- Docker

1.1 NODE.JS

Node.js è un linguaggio multiplatforma orientato agli eventi (*event-based*) per l'interpretazione di codice JavaScript.

Inizialmente JavaScript veniva utilizzato unicamente lato client con del codice incorporato in documenti *html*, eseguito poi da un interprete presente nativamente nei browser web. Con l'utilizzo di **Node.js** è possibile utilizzare JS anche per eseguire codice lato server, ad esempio per l'invio di file (html, css, json, xml, js, ...). In questo modo si implementa il paradigma “*javascript everywhere*” unificando lo sviluppo di applicazioni web intorno ad un unico linguaggio.

Node.js ha un'architettura orientata agli eventi, il che rende possibile l'**I/O asincrono** che permette di ottimizzare la scalabilità delle applicazioni web con molte operazioni di input/output (ottimo per applicazioni web real-time).

Per la realizzazione di quest'elaborato è stato creato un web server basato sul meccanismo delle **restAPI**, mediante l'utilizzo del framework minimale per Node.JS, **Express**. Esso fornisce un vasto set di funzionalità

per lo sviluppo di applicazioni web e mobile e facilita lo sviluppo di applicazioni web basate su Node.

```
19  var app = require('express')();
20  const https = require('https');
21
22  const options = {
23    key: fs.readFileSync('certs/private.key'),
24    cert: fs.readFileSync('certs/certificate.crt'),
25    passphrase: process.env.CERT
26  };
```

Figura 1.1.1

Sono stati utilizzati certificati auto-firmati per l'utilizzo di **HTTPS**.

```
138  // Server
139  const server = https.createServer(options, app);
140
141  server.listen(port, () => {
142    console.log('[SERVER] Listening on https://' + ip.address() + ':' + port);
143  });
```

Figura 1.1.2

1.2 JANUS

Server WebRTC general purpose, fornisce unicamente le funzionalità per l'implementazione dei mezzi che consentono di stabilire una comunicazione WebRTC con un browser web, lo scambio di messaggi JSON e l'inoltro di pacchetti RTP, RTCP e messaggi tra il browser ed il server. Qualsiasi caratteristica specifica è fornita attraverso un meccanismo di *plug-in* lato server (i browser possono quindi contattare Janus per sfruttare le funzionalità fornite). Per lo sviluppo dell'elaborato è stato utilizzato il plug-in *VideoRoom*, il quale ha facilitato l'implementazione e la realizzazione di una video conferenza.

Janus è stato opportunamente configurato per operare attraverso **HTTPS** mediante, anche in questo caso, dei certificati auto-firmati. Per fare ciò è stato necessario modificare i file *janus.jcfg*, *janus.plugin.videoroom.jcfg* e *janus.transport.http.jcfg*.

Nel primo file sono stati configurati i server **stun.stunprotocol.org** (STUN) e **turn.anyfirewall.com** (TURN).

```
nat: {  
    stun_server = "stun.stunprotocol.org"  
    stun_port = 3478  
    nice_debug = true  
    full_trickle = true  
    ice_nomination = "regular"  
    ice_keepalive_conncheck = true  
    ice_lite = true  
    ice_tcp = true
```

Figura 1.2.1: configurazione server STUN

```
turn_server = "turn.anyfirewall.com"  
turn_port = 443  
turn_type = "tcp"  
turn_user = "webrtc"  
turn_pwd = "webrtc"
```

Figura 1.2.2: configurazione server TURN

Nel secondo file sono state create due *room* di default per le video conferenze.

```

room-1234: {
  description = "Unina Room 1"
  secret = "adminpwd"
  publishers = 6
  bitrate = 128000
  fir_freq = 10
  #audiocodec = "opus"
  #videocodec = "vp8"
  record = false
  #rec_dir = "/path/to/recordings-folder"
}

room-1235: {
  description = "Unina Room 2"
  secret = "adminpwd"
  publisher = 6
  bitrate = 128000
  fir_freq = 10
  record = false
}

```

Figura 1.2.3: room di default

Nel terzo, ed ultimo, sono stati impostati i path da cui Janus legge i certificati per l'utilizzo di **HTTPS**.

```

certificates: {
  cert_pem = "/certs/certificate.crt"
  cert_key = "/certs/private.key"
  cert_pwd = "uninawebrtc"
  #ciphers = "PFS:-VERS-TLS1.0:-VERS-TLS1.1:-3DES-CBC:-ARCFOUR-128"
}

```

Figura 1.2.4: configurazione di https

1.3 DOCKER

Docker è un progetto *open-source* atto ad automatizzare il processo di sviluppo delle applicazioni all'interno di *container* software, fornendo un'astrazione aggiuntiva grazie alla virtualizzazione a livello di sistema operativo di Linux.

Docker utilizza le funzioni di isolamento delle risorse del Kernel Linux (ad esempio *cgroup* e *namespace*) per consentire a *container* indipendenti di coesistere sulla stessa istanza di quest'ultimo, evitando così l'installazione e la configurazione di una macchina virtuale. I *namespace* per lo più isolano ciò che l'applicazione può vedere dell'ambiente

operativo, mentre i **cgroup** consentono l'isolamento delle risorse, inclusa la CPU, la memoria, i dispositivi di I/O a blocchi e la rete.

Per lo sviluppo dell'elaborato sono state utilizzate tre immagini Docker:

- **ale13/janus-gateway:v5** – basata su un'immagine Ubuntu su cui è stato installato Janus grazie alla guida ufficiale presente su GitHub.
- **miscia/node-server:v5** – immagine contenente il back-end dell'applicazione. Creata mediante il seguente **Dockerfile**:

```
Dockerfile > ...
1  FROM node:16
2
3  # Create app directory
4  WORKDIR /usr/src/app
5
6  COPY package*.json ./
7
8  RUN npm install
9  COPY . .
10 EXPOSE 8008
11
12 CMD [ "npm", "start" ]
13
```

Figura 1.3.1: Dockerfile

- **mongo**: immagine ufficiale del database non relazionale mongoDB. È stato creato, in oltre, lo script **init-mongo.js** per la configurazione del database.

```

JS init-mongo.js > ...
1  db.createUser({
2      user: 'root',
3      pwd: 'pippozzo',
4      roles: [{
5          role: 'readWrite',
6          db: 'WebRTC'
7      }]
8  });
9
10 db = new Mongo().getDB('WebRTC');
11
12 db.createCollection('WebRTC');

```

Figura 1.3.2: init-mongo.js

2 ARCHITETTURA

Per la realizzazione di *Unina Video Conference* si è scelto di adottare un'architettura a micro-servizi, costituita da:

- *Server*
- *Janus-Gateway*
- *Database*

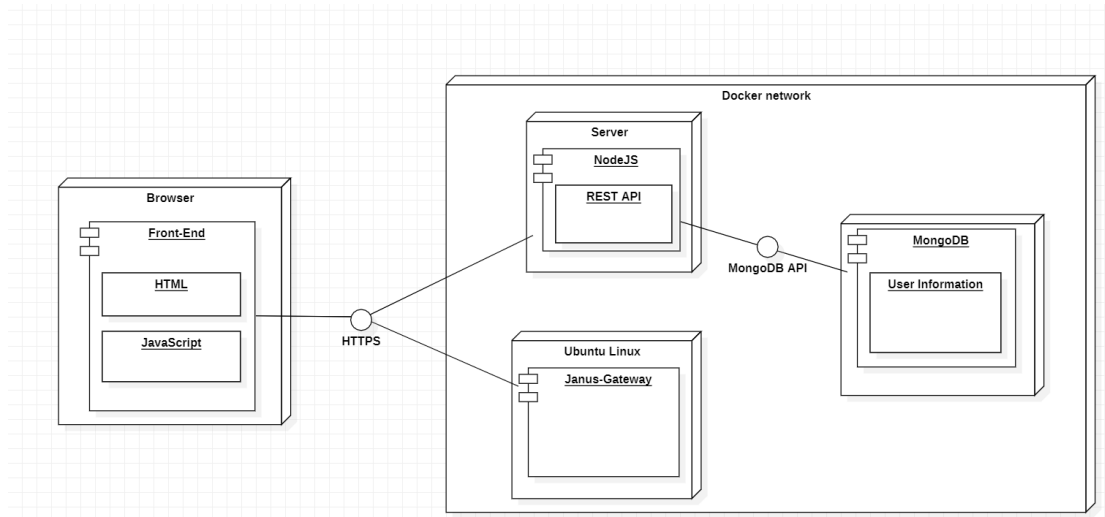


Figura 2.1: Deployment Diagram

Per la distribuzione dell'intero progetto, i tre micro-servizi sono stati *containerizzati* ed è stata configurata una sottorete virtuale mediante l'utilizzo del file ***docker-compose.yml***.

Per comprendere al meglio il funzionamento di quest'ultimo è possibile analizzarlo dividendolo in due parti: ***Networks*** e ***Services***.

Nella prima parte sono stati definiti i micro-servizi.

```

2  services:
3
4  mongo:
5    networks:
6      network_1:
7        ipv4_address: 194.20.1.3
8    image: 'mongo'
9    container_name : 'mongodb'
10   environment:
11     - MONGO_INITDB_DATABASE=WebRTC
12     - MONGO_INITDB_ROOT_USERNAME=database_wrtc_name
13     - MONGO_INITDB_ROOT_PASSWORD=database_wrtc_pass
14
15   volumes:
16     - ./init-mongo.js:/docker-entrypoint-initdb.d/init-mongo.js:ro
17   ports:
18     - '27017-27019:27017-27019'
19   restart: always
20
21  server:
22    networks:
23      network_1:
24        ipv4_address: 194.20.1.4
25    image: miscia/node-server:v5
26    container_name : server_WebRTC
27    environment:
28      - MONGO=mongodb://root:pipposso@194.20.1.3:27017/WebRTC
29      - USER= [EMAIL]
30      - PASS= [PASSWORD]
31      - JWT=jldtbo8eelemaprkd
32      - CERT=uninawebrtc
33    ports:
34      - '8008:8008'
35    restart: always
36
37  janus:
38    networks:
39      network_1:
40        ipv4_address: 194.20.1.5
41    image: ale13/janus-gateway:v5
42    container_name : janus_gateway
43    command: /opt/janus/bin/janus
44    ports:
45      - '8088-8089:8088-8089'
46    restart: always

```

Figura 2.2: Services

Attraverso questa sezione è stato possibile configurare:

- Le immagini Docker da utilizzare.
- I nomi dei container.
- Gli indirizzi IP dei container.
- Le porte esposte.
- Eventuali variabili d'ambiente.
- Eventuali comandi da eseguire all'avvio.

Nella seconda parte, invece, è stata configurata la rete virtuale utilizzata dai container.

```
48 networks:
49     network_1:
50         ipam:
51             config:
52                 - subnet: 194.20.1.1/24
```

Figura 2.3: Networks

2.1 SERVER

Web Server interamente scritto in JavaScript mediante l'utilizzo del modulo **express**, il quale gestisce le richieste dei client attraverso il meccanismo di **restAPI**, ovvero:

“interfacce di programmazione che usano HTTP per gestire dati remoti”

Nella *Figura 4.1.1* vi è un esempio di come vengono gestite le richieste GET e POST da parte del server.

```

83 app.get('/videoroom', (req, res) => {
84   const token = req.cookies["session"];
85   // console.log('[DEBUG] Token: ' + token);
86   var status_code = verify(token);
87   if (status_code == 200) {
88     console.log('[VIDEOROOM] ' + Date.now() + ' - GET - ' + req.hostname);
89     res.sendFile(__dirname + '/client/videoroom.html');
90   } else {
91     var room = req.query.room;
92     if (room !== undefined) {
93       res.redirect('/login?room=' + room);
94     } else {
95       res.redirect('/login');
96     }
97   }
98 });
99
100 //POST METHOD
101 app.post('/login', (request, response) => {
102   console.log('[LOGIN] ' + Date.now() + ' - POST - ' + request.hostname);
103   db.login(request.body, (token, res, status) => {
104     switch (status){
105       case 200:
106         response.status(status);
107         response.cookie('session', token);
108         if(request.body.room === "") {
109           response.redirect('/videoroom');
110         } else {
111           response.redirect('/videoroom?room=' + request.body.room);
112         }
113         break;
114       default:
115         response.status(status).end();
116     }
117   });
118 });

```

Figura 2.1.1: esempio di API REST

In questo scenario, il server si occupa di: inviare file *.html* e *.js* che vengono richiesti dai client, effettuare controlli di sicurezza come *Access Control* ed *Autenticazione*, di gestire lo scambio di dati con il database.

Inoltre, tutte le comunicazioni col server avvengono mediante **HTTPS** grazie all'utilizzo di una chiave privata e ad un certificato *self-signed*.

2.2 JANUS-GATEWAY

La comunicazione col server Janus avviene lato client mediante l'utilizzo di **HTTPS** (sulla porta 8089) attraverso la tecnica del *Long Polling*. Essa consiste nell'invio di una richiesta da parte del client verso il server con l'unico scopo di aprire una connessione e mettersi in attesa di una risposta, non appena il server vorrà contattare l'end-point verrà inviata una risposta a

quest'ultimo il quale, potrà riaprire la connessione attraverso un'ulteriore richiesta oppure chiudere la connessione.

Janus implementa **WebRTC**, grazie al quale, si occupa di stabilire una connessione *peer-to-peer* attraverso l'uso dei protocolli:

- **JSEP & SDP**: utilizzati per la fase di *Signalling* e di negoziazione.
- **ICE**: utilizzato per aggirare il problema del *NAT-Traversal* attraverso l'utilizzo del protocollo **STUN** e della sua estensione **TURN**.
- **RTP/RTCP - SRTP - DTLS**: per lo scambio sicuro dello streaming multimediale (Audio, Video, etc...).
- **SCTP**: per scambio di dati generici.

2.3 DATABASE

Per il salvataggio delle credenziali di accesso degli utenti è stato utilizzato il database non relazionale **MongoDB**. Esso comunica unicamente col server attraverso il modulo **mongodb.js** in cui sono state scritte le funzioni per la creazione di query al database.

```

17 MongoClient.connect(connectionString, { useUnifiedTopology: true }).then(
18   (client) => {
19     console.log("[MONGODB] Connected to Database");
20
21     db = client.db("WebRTC");
22     usersCollection = db.collection("WebRTC");
23     // scheduleCollection = db.collection("schedule");
24
25     clientMongo = client;
26   });
27
28
29 // LOGIN AND REGISTRATION
30
31 > function isEmailUnique(email, callback) {--
32   ...
33 }
34
35
36
37
38
39
40 module.exports = {
41   // LOGIN AND REGISTRATION
42
43   insertUser: function (user, callback) {
44
45     isEmailUnique(user.email, function (res) {
46       if (res.length == 0) {
47         bcrypt.cryptPassword(user.password, function (err, hash) {
48           if (err) throw err;
49
50           var myobj = {
51             name: user.name,
52             surname: user.surname,
53             email: user.email,
54             password: hash,
55             confirmed: false
56           };
57
58           usersCollection.insertOne(myobj, function (err) {
59             if (err) throw err;
60             nodemailer.sendConfirmationEmail(
61               user.name,
62               user.email
63             );
64             callback(true);
65           });
66         });

```


Figura 2.3.1: mongodb.js

3 FUNZIONAMENTO

L'applicazione nasce con l'obiettivo di realizzare una videoconferenza in tempo reale tra i partecipanti che si connettono nell'ambito di una stessa room, con la possibilità di accedere alla stanza di default offerta dell'applicazione o di poter realizzare la propria room.

3.1 ISCRIZIONE E LOGIN

L'applicazione realizzata prevede una prima fase in cui l'utente, per usufruire del servizio offerto, dovrà effettuare la registrazione presso la pagina iniziale fornendo una serie di informazioni personali:



Create a new account.

Name

Surname

Email

Password

Password (repeat)

Sign Up

You already have an account? [Login here.](#)

Figura 3.1.1

Viene richiesto il nome e cognome della persona, email ed infine una password che abbia almeno 8 caratteri; ciascun campo della form deve essere compilato; in caso di mancata compilazione di uno dei campi, non sarà possibile effettuare la registrazione e quindi accedere al servizio.

Non appena ci si registra, se tutto è andato a buon fine comparirà una schermata per avvisare l'utente dell'avvenuta registrazione e della necessaria conferma da effettuare mediante email:

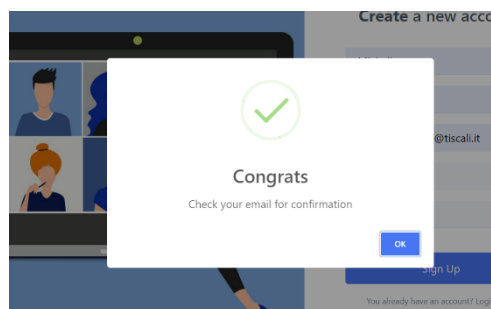


Figura 3.1.2

Pertanto, l'utente troverà nella propria casella di posta una email inviata dall'applicazione per garantire l'avvenuta conferma. Fatto ciò, l'applicazione provvederà a reindirizzare l'utente alla pagina di Login dell'applicazione:

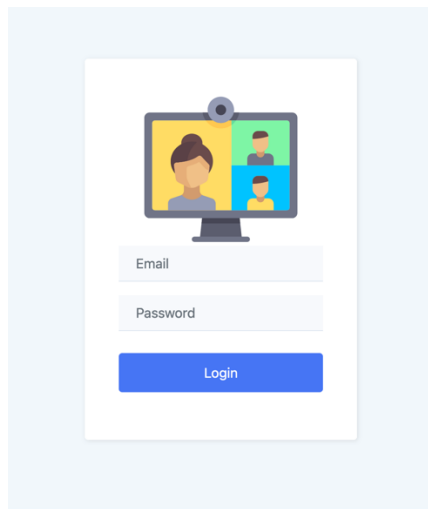


Figura 3.1.3: schermata di Login

Nell'eventualità l'utente non inserisca correttamente i campi Email e Password gli sarà negato l'accesso all'applicazione.

3.2 START

Dopo aver effettuato il Login l'utente si troverà davanti la seguente schermata:



Figura 3.2.1

Intuitivamente l'utente potrà premere il tasto Start per proseguire nella fruizione del servizio, o eventualmente effettuare il logout dall'applicazione con il tasto apposito.

Continuando nella navigazione dell'applicazione sarà possibile, come anticipato, la creazione di una nuova room, o la possibilità di prendere parte ad una room già esistente di default o ad una room creata da un altro partecipante:



Figura 3.2.2

3.3 VIDEOROOM

La schermata presente in *Figura 3.3.1* mostra l'interfaccia della videoroom realizzata:

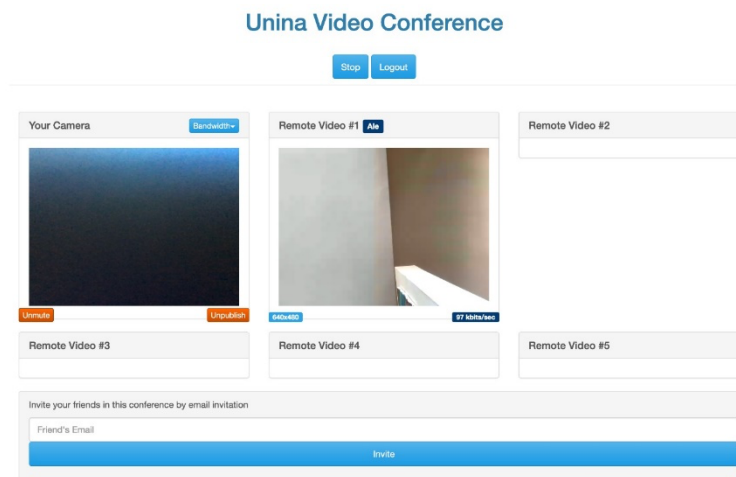


Figura 3.3.1

si hanno a disposizione 6 quadranti, ciascuno per ogni partecipante alla riunione.

Un'opzione aggiuntiva è la possibilità di invitare altri partecipanti a prenderne parte; ciò viene fatto attraverso il tasto *Invite* affiancato da una form dove verrà ripotata la email della persona da voler invitare. Fatto ciò, il nostro invitato riceverà nella propria casella di posta elettronica l'invito ufficiale a prendere parte alla riunione.

La persona invitata, se già in possesso di un Account, avrà solo bisogno di effettuare l'accesso e sarà automaticamente reindirizzato nella room con gli altri partecipanti; in caso contrario, avrà bisogno innanzitutto di creare un proprio account mediante Registrazione e successivamente il Login.

3.4 CASI D'USO

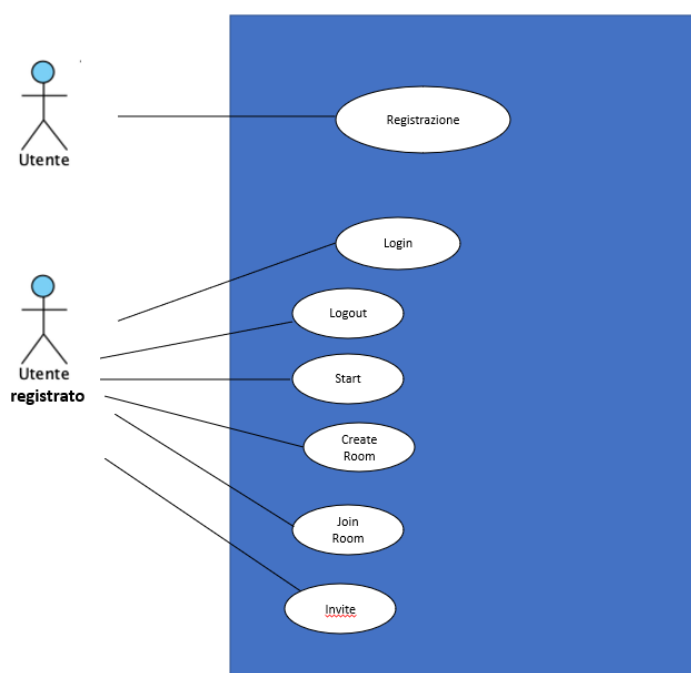


Figura 5.4.1

Attori :

- Utente
- Utente-Registrato

Casi D'Uso – Breve Descrizione •

- Registrazione: permette all'utente di accedere alla web-app

- Login: permette all'utente registrato di accedere alla propria area privata •
- Logout: una volta effettuato il login, permette di disconnettersi successivamente.
- Start: dà la possibilità all'utente di accedere all'area principale dell'applicazione.
- Create Room: permette all'utente di creare una room in cui avviare la propria videoconferenza
- Join room: permette all'utente di inserire il codice di una room già esistente e prendere parte alla conferenza
- Invite: comando che consente di invitare altri partecipanti mediante il loro indirizzo email.

4 JANUS

Come anticipato in precedenza, la video conferenza è stata implementata sfruttando il plugin *Videoroom*. La comunicazione con quest'ultimo e in generale con il server Janus è realizzata all'interno dello script ***video.js***, contenuto all'interno della pagina ***videoroom.html*** tramite la quale il client può interagire con tutte le funzionalità relative al plugin.

Di seguito è riportato un sequence diagram che evidenzia le varie interazioni con il plugin videoroom nel caso in cui si voglia entrare in una stanza:

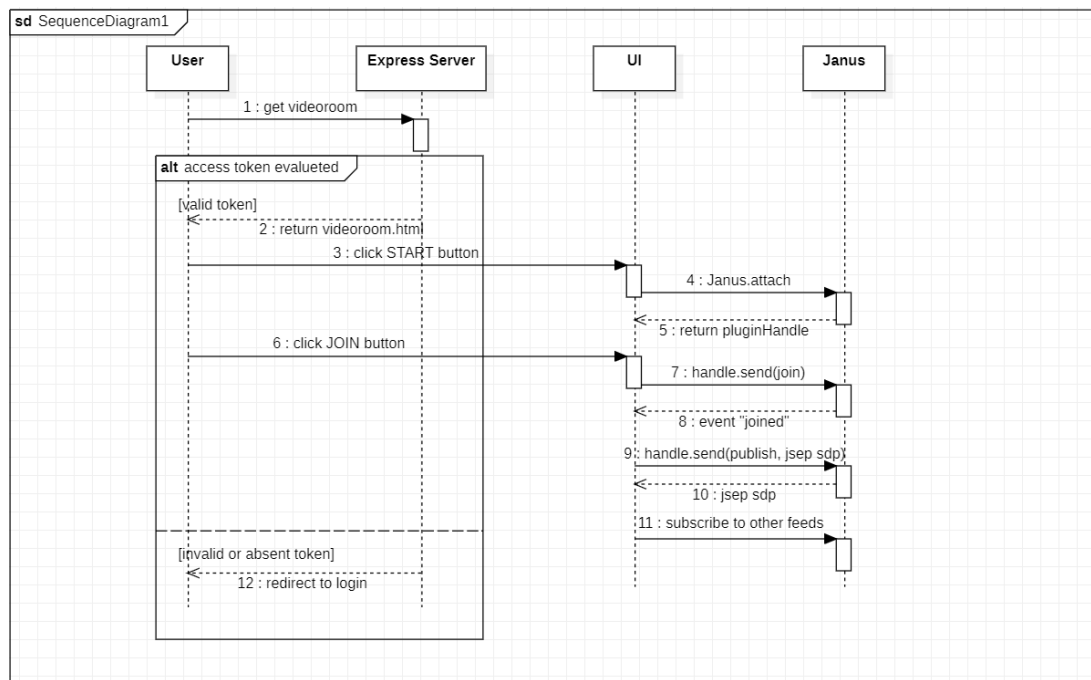


Figura 4.1

4.1 CONNESSIONE AL PLUGIN (PUBLISHER)

La connessione con il server viene avviata dopo la pressione del pulsante *start*. L'API di Janus permette quindi di associare una funzione di callback allo stato di successo di questa connessione. All'interno di quest'ultima si utilizza la funzione ***Janus.attach*** per realizzare la connessione con il plugin *videoroom*. Il valore di ritorno di questa funzione è un *pluginHandle*, oggetto che permette la comunicazione con il plugin e che verrà utilizzato nella callback relativa allo stato di successo della connessione con esso.

```

janus = new Janus(
{
  server: server,
  success: function() {
    // Attach to VideoRoom plugin
    janus.attach(
    {
      plugin: "janus.plugin.videoroom",
      opaqueId: opaqueId,
      success: function(pluginHandle) {
        // Attach to VideoRoom plugin
      }
    }
  )
}
)
  
```

Figura 4.1.1

È bene notare che il plugin *videoroom* è basato su un pattern del tipo *publish/subscribe*: ogni partecipante ad una room può pubblicare il proprio feed e ricevere una lista di altri feed a cui iscriversi. Questa prima connessione viene quindi utilizzata per la fase di *publishing*; successivamente verrà realizzata una nuova connessione per ogni feed al quale un utente si iscrive.

4.2 CREATE AND JOIN ROOM

Nel momento in cui la connessione con il plugin va a buon fine l'utente può scegliere, mediante la pressione di due diversi pulsanti, di creare una nuova stanza o di unirsi ad una già esistente. Entrambe queste funzionalità vengono realizzate sfruttando il *pluginHandle* per inviare dei messaggi in formato JSON opportunamente costruiti.

```
function createRoom() {
  new_room_name = parseInt($('#roomid').val());
  var json_req = {
    request: 'create',
    room: new_room_name,
    description: "Unina Room",
    secret: "adminpwd",
    publishers: 6,
    bitrate: 128000,
    fir_freq: 10,
    record: false
  }
  $('#create').addClass('hide').hide();
  $('#join').removeClass('hide').show();

  sfutest.send({ message: json_req });
}
```

Figura 4.2.1: *createRoom()*

Questo tipo di richiesta permetterà di creare una stanza che avrà un identificativo contenuto nella variabile *new_room_name* e scelto dall'utente in fase di creazione, un segreto necessario per l'edit o la

cancellazione della stanza e ulteriori informazioni necessarie per la gestione della video-conferenza quali ad esempio il numero massimo di publishers, il bitrate e la possibilità di registrare la chiamata.

Mediante lo stesso meccanismo è possibile creare una richiesta per entrare all'interno di una stanza:

```
var register = {
  request: "join",
  room: myroom,
  ptype: "publisher",
  display: username
};
$('#join').addClass('hide').hide();
myusername = escapeXmlTags(username);
sfutest.send({ message: register });
```

Figura 4.2.2

In questo caso la richiesta sarà di tipo *join*, viene indicata la stanza nella quale si vuole entrare e il nome che verrà mostrato agli altri utenti. Il parametro *ptype* permette invece di realizzare la distinzione tra publishers e subscribers.

In questa implementazione il parametro *display* coincide con il nome scelto dall'utente in fase di registrazione e viene ottenuto analizzando i *JWT* utilizzati per l'autenticazione:

```
function getUsername() {
  const token = document.cookie.split('; ').find(row => row.startsWith('session=')).split('=')[1];
  var decoded = parseJwt(token);
  var username = decoded['name'];
  return username;
}
```

Figura 4.2.3: getUsername()

Dopo aver inviato queste richieste bisognerà gestire opportunamente le risposte del server Janus per pubblicare il proprio video ed iscriversi ai feed degli altri partecipanti.

4.3 GESTIONE RISPOSTE DEL SERVER

La funzione *janus.attach* permette di specificare una funzione di callback che verrà eseguita in corrispondenza dell'evento *onmessage*, che rappresenta l'arrivo di un messaggio da parte di Janus. I messaggi di risposta sono anch'essi in formato JSON e presentano all'interno del campo *videoroom* l'evento a cui essi si riferiscono. Di particolare importanza è la gestione dell'evento "*joined*", ottenuto in risposta ad una richiesta di *join*:

```
}else if(event === "joined") {  
    myid = msg["id"];  
    mypvtid = msg["private_id"];  
    Janus.log("Successfully joined room " + msg["room"] + " with ID " + myid);  
    if(subscriber_mode) {  
        $('#videojoin').hide();  
        $('#videos').removeClass('hide').show();  
    } else {  
        publishOwnFeed(true);  
    }  
    // Any new feed to attach to?  
    if(msg["publishers"]) {  
        var list = msg["publishers"];  
        Janus.debug("Got a list of available publishers/feeds:", list);  
        for(var f in list) {  
            var id = list[f]["id"];  
            var streams = list[f]["streams"];  
            for(var i in streams) {  
                var stream = streams[i];  
                stream["id"] = id;  
                stream["display"] = display;  
            }  
            feedStreams[id] = streams;  
            Janus.debug(" >> [" + id + "] " + display + ":", streams);  
            newRemoteFeed(id, display, streams);  
        }  
    }  
}
```

Figura 4.3.1

La prima funzione che viene eseguita è *publishOwnFeed*, tramite la quale sarà possibile negoziare definitivamente una *peer connection* con il plugin videoroom e pubblicare quindi il proprio feed. Nel caso in cui ci siano già altri utenti all'interno della stanza, Janus invierà nel messaggio *joined* anche la lista di feeds al quale è possibile iscriversi. Per ognuno di questi

verrà eseguita la funzione *newRemoteFeed*, tramite la quale è possibile creare nuove peer connection come *subscriber*.

È fondamentale anche la gestione dell'evento *event*, tramite il quale Janus notifica i publishers presenti all'interno di una stanza dell'aggiunta di un nuovo feed a cui potersi iscrivere. Anche in questo caso verrà quindi chiamata la funzione *newRemoteFeed* per la gestione dell'iscrizione:

```
} else if(event === "event") {  
    // Any info on our streams or a new feed to attach to?  
    if(msg["streams"]) {  
        var streams = msg["streams"];  
        for(var i in streams) {  
            var stream = streams[i];  
            stream["id"] = myid;  
            stream["display"] = myusername;  
        }  
        feedStreams[myid] = streams;  
    } else if(msg["publishers"]) {  
        var list = msg["publishers"];  
        Janus.debug("Got a list of available publishers/feeds:", list);  
        for(var f in list) {  
            var id = list[f]["id"];  
            var display = list[f]["display"];  
            var streams = list[f]["streams"];  
            for(var i in streams) {  
                var stream = streams[i];  
                stream["id"] = id;  
                stream["display"] = display;  
            }  
            feedStreams[id] = streams;  
            Janus.debug("  >> [" + id + "] " + display + ":", streams);  
            newRemoteFeed(id, display, streams);  
        }  
    }  
}
```

Figura 4.3.2

4.4 PUBLISHOWNFEED

Come anticipato in precedenza, lo scopo di questa funzione è quello di negoziare una peer connection con Janus, in modo da rendere il publisher attivo a tutti gli effetti.

Come prima cosa viene eseguita, tramite il *pluginHandle*, la funzione *createOffer*, nella quale vengono specificati i media a cui siamo interessati

(audio e video) e se si vuole inviare e/o ricevere (in questo caso si è interessati al solo invio poiché viene gestita la connessione di un *publisher*). Nel caso in cui questa funzione vada a buon fine, si otterrà un messaggio SDP che potrà essere inviato a Janus all'interno di un messaggio di tipo *publish*. La risposta a questo messaggio sarà l'SDP proveniente da Janus; una volta ricevuto quest'ultimo, la negoziazione della peer connection sarà completa.

```
function publishOwnFeed(useAudio) {
  // Publish our stream
  $('#publish').attr('disabled', true).unbind('click');
  sfutest.createOffer(
    [
      media: { audioRecv: false, videoRecv: false, audioSend: useAudio, videoSend: true },
      simulcast: doSimulcast,
      customizeSdp: function(jsep) {
        if(doDtx) {
          jsep.sdp = jsep.sdp
            .replace("useinbandfec=1", "useinbandfec=1;usedtx=1")
        }
      },
      success: function(jsep) {
        Janus.debug("Got publisher SDP!", jsep);
        var publish = { request: "configure", audio: useAudio, video: true };
        if(acodec)
          publish["audiocodec"] = acodec;
        if(vcodec)
          publish["videocodec"] = vcodec;
        sfutest.send({ message: publish, jsep: jsep });
      }
    ]
  );
}
```

Figura 4.4.1

4.5 NEWREMOTEFEED

L'iscrizione ai feed dei partecipanti di una stanza avviene allo stesso modo della connessione come publisher. Di conseguenza, verrà eseguita la funzione *janus.attach* per negoziare una nuova connessione al plugin per ogni feed a cui ci si iscrive. Anche in questo caso sarà quindi necessario inviare un messaggio di tipo *join* per ogni iscrizione:

```
var subscribe = {  
  request: "join",  
  room: myroom,  
  ptype: "subscriber",  
  streams: subscription,  
  private_id: mypvtid  
};  
remoteFeed.send({ message: subscribe });
```

Figura 4.5.1

In questo caso il parametro *ptype* sarà uguale a *subscriber* e sarà inoltre presente una lista dei feed a cui ci si vuole iscrivere. La risposta a questo messaggio sarà di tipo *attached*.

Una volta completata l'iscrizione, all'interno dell'evento *onremotetrack*, il quale specifica la disponibilità di una *MediaStreamTrack* remota, sarà possibile specificare una funzione di callback che permette di modificare opportunamente la pagina *videoroom.html* per mostrare i feed di tutti i publisher presenti all'interno della stanza.

5 SICUREZZA

La sicurezza ha un'importanza fondamentale in quanto è necessaria per garantire la disponibilità, l'integrità e la riservatezza delle informazioni. Per la realizzazione dell'elaborato si è scelto di soffermarsi su alcuni aspetti in particolare inerenti alla sicurezza.

5.1 ACCESS CONTROL

La politica e le procedure di controllo degli accessi riguardano i controlli nella famiglia AC implementati all'interno di sistemi e organizzazioni. Per l'implementazione delle politiche di *Access Control* sono state seguite le linee guida forniteci dal NIST.

Per la gestione delle sessioni e soprattutto, per evitare che utenti non autorizzati potessero accedere a pagine sensibili, è stato utilizzato un **Json Web Token**.

```
74 login : function(user, callback){
75
76   var query = {email: user.email};
77
78   usersCollection.find(query).toArray(function (err, res) {
79
80     if (err) throw err; //TODO
81
82     if(res.length == 0) callback(null, null, 404);
83     else{
84       if(res[0].confirmed == true){
85         bcrypt.comparePassword(user.password, res[0].password, function(err, isPasswordMatch){
86           if(err) throw err; //TODO
87           if(isPasswordMatch){
88             const token = jwt.sign({_id: res[0]._id, name: res[0].name}, process.env.JWT, {expiresIn: "30m"});
89
90             callback(token, res[0], 200);
91           } else callback(null, null, 401);
92         })
93       } else {
94         //Email not confirmed
95         callback(null, null, 401);
96       }
97     }
98   });
99 },
100 }
```

Figura 5.1.1: assegnazione del JWT

Durante la fase di *login* viene generato un JWT, con una validità di 30 minuti, contenente l'id ed il nome dell'utente che sta effettuando l'accesso, dopodiché verrà salvato lato client attraverso la creazione di un cookie chiamato *session*.

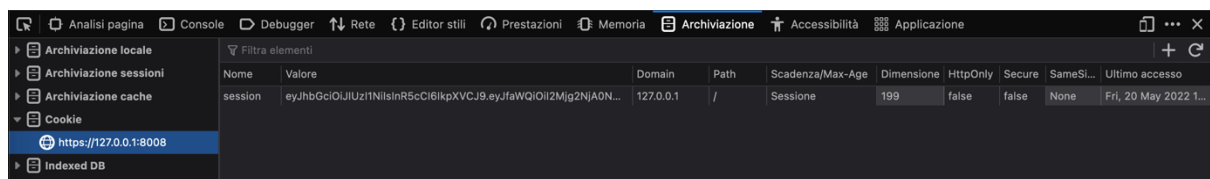


Figura 5.1.2: session cookie

Il controllo sulla validità del token viene effettuato mediante il file *verifyToken.js*, il quale viene richiamato dal server in ogni API.

```

1  const jwt = require('jsonwebtoken');
2
3  module.exports = function (token){
4
5      if(!token) {
6          return 401;
7      }
8
9      try{
10         const verified = jwt.verify(token, process.env.JWT);
11         return 200;
12     }catch(err){
13         console.log('[ERROR] ' + err);
14         return 401;
15     }
16 }
17

```

Figura 5.1.3: *verifyToken.js*

5.2 ENVIRONMENT VARIABLES

Per la realizzazione di codice sicuro si è evitato di inserire dati sensibili in chiaro all'interno dei file. Pertanto, si è scelto di utilizzare un container con delle variabili d'ambiente, totalmente configurabili all'interno del file *docker-compose.yml*. Dopodiché, sono state richiamate opportunamente nel codice affinché venissero interpretate correttamente (è possibile osservarne un esempio nella *Figura 6.1.3* alla linea 10 del codice).

5.3 FILTRAGGIO E VALIDAZIONE

Per evitare attacchi di tipo *Injection* e *Cross Site Scripting* si è scelto non lasciare un'eccessiva libertà agli utenti per la creazione degli input da inviare al server. Difatti sono state utilizzate delle *regular expression* per evitare l'inserimento di caratteri “pericolosi” (<, >, “, ‘, &, etc...) nelle form presenti nell'applicazione. È possibile vederne un'esempio nella *Figura 6.3.1*.

```

68     function validateText(name){
69         var regName = new RegExp("[a-zA-Z]+");
70
71         if(!regName.test(name)){
72             return false;
73         }
74         return true;
75     }

```

Figura 5.3.1: regular expression presente nella fase di registrazione.

5.4 EMAIL CONFIRMATION

Un ulteriore controllo di sicurezza, è stato il controllo della validità dell'email, tramite una mail di conferma inviata a un cliente dopo l'attivazione di una determinata condizione.

Nel caso specifico dell'elaboraato tale email viene inviata non appena la richiesta di registrazione viene effettuata; l'utente riceverà nella sua casella di posta l'apposita email inviata dal provider del servizio per verificare la sua identità:

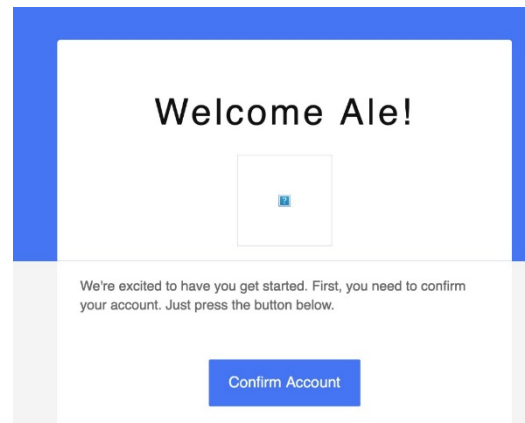


Figura 5.3.2: email di conferma

Non appena l'utente clicca sul tasto 'Confirm', in background ciò che accade è che all'interno del database MongoDB, la entry associata

all'utente, venutasi a creare in fase di Sign Up per la registrazione, sarà confermata:



Figura 5.3.3: MongoDB confirmation

Ciò che invece accade nel browser sarà l'inoltro alla pagina di Login in cui l'utente potrà finalmente accedere al servizio.