**Ministerul Educaţiei al Republicii Moldova**

**Universitatea Tehnică A Moldovei**

**Facultatea Calculatoare, Informatică Şi Microelectronică**

**Departamentul Ingineria Software și Automatică**

**Raport**

la lucrarea de laborator

**Disciplina:** „Programarea aplicaţiilor mobile”

**Tema: „Agent de mesagerie - Message Broker”**

**A ELABORAT:**

**Student: gr.TI-171** (Musteața Dorin)

Grupa academică, Numele, Prenumele

**A VERIFICAT:**

**\_\_\_\_\_\_\_\_\_\_\_\_**(lector univ.mag., Antohi Ionel)

**Chişinău – 2021**

**Cuprins**

[**Scopul lucrării de laborator** 3](#_Toc88160467)

[**Structura proiectului** 4](#_Toc88160468)

[**Codul sursă** 5](#_Toc88160469)

[**auth\_\_index.js** 5](#_Toc88160470)

[**auth\_\_passport.js** 6](#_Toc88160471)

[**models\_\_Conversation.js** 6](#_Toc88160472)

[**models\_\_Message.js** 7](#_Toc88160473)

[**models\_\_User.js** 8](#_Toc88160474)

[**index.js** 9](#_Toc88160475)

[**.env** 15](#_Toc88160476)

[**Rezultate** 16](#_Toc88160477)

[**Concluzie** 28](#_Toc88160478)

# **Scopul lucrării de laborator**

Integrarea bazată pe agenți de mesaje care ar permite o comunicare asincronă dintre componentele distribuite ale unui sistem.

**Obiectivele specifice ale lucrării**

1. Definirea protocolului de lucru al agentului de mesaje [1]

a. Formatul (tipul) mesajelor de transmis. Se recomandă utilizarea formatului XML

b. Numărului de canale unidirecționale (variabil/fix, dependent de tipul mesajelor, etc.)

c. Structura comunicației asigurată de agent (unul-la-unu sau unul-la-mulți)

d. Politici de livrare pentru diverse cazuri definite de logica de lucru al agentului (mesaje invalide, căderea agentului, etc.)

2. Elaborarea nivelului abstract de comunicare (rețea) necesară elementelor pentru primirea/transmiterea mesajelor de către emițător-agent-receptor;

a. Protocolul de transport se alege în dependență de obiectivele protocolului de lucru

b. Tratarea concurentă a cererilor

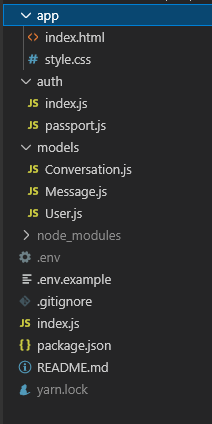
3. Elaborarea elementelor ce asigură păstrarea mesajelor primite

a. Metoda transientă: mesajele vor fi stocate în colecții concurente de date specifice limbajului selectat

b. Metoda persistentă: mesajele vor fi serializate/deserializate utilizînd metode de procesare asincronă sau concurentă

4. Elaborarea nivelului abstract de rutare a mesajelor.

# **Structura proiectului**



Proiectul a fost elaborat în mediul Node.js v14 , care implementează o baza de date MongoDB , un message broker RabbitMQ , protocolul websockets și autentificarea JWT.

Proiectul poate fi accesat folosind github: <https://github.com/fcim-dm/rabbitmq-node>

# **Codul sursă**

## **auth\_\_index.js**

|  |
| --- |
| const passport = require("passport");  const jwt = require("jsonwebtoken");  const { ExtractJwt } = require("passport-jwt");  const jwtOptions = {    secretOrKey: process.env.JWT\_SECRET,    jwtFromRequest: ExtractJwt.fromAuthHeaderWithScheme("Bearer"),  };  const handleJWT = (req, res, next) => async (err, user, info) => {    const error = err || info;    try {      const logIn = req.logIn;      if (error || !user) throw error;      await logIn(user, {        session: false,      });    } catch (e) {      return next(e);    }    req.user = user;    return next();  };  exports.authorize = () => (req, res, next) => {    // check for token in query    if (req.query.token) {      req.headers.authorization = "Bearer " + req.query.token;      req.headers["content-type"] = "application/json";    }    // auth passport    passport.authenticate(      "jwt",      {        session: false,      },      handleJWT(req, res, next)    )(req, res, next);  }; |

## **auth\_\_passport.js**

|  |
| --- |
| const JwtStrategy = require("passport-jwt").Strategy;  const { ExtractJwt } = require("passport-jwt");  const User = require("../models/User");  const jwtOptions = {    secretOrKey: process.env.JWT\_SECRET,    jwtFromRequest: ExtractJwt.fromAuthHeaderWithScheme("Bearer"),  };  const jwt = async (payload, done) => {    try {      const user = await User.findById(payload.\_id);      if (user) return done(null, user);      return done(null, false);    } catch (error) {      return done(error, false);    }  };  exports.jwt = new JwtStrategy(jwtOptions, jwt); |

## **models\_\_Conversation.js**

|  |
| --- |
| const mongoose = require("mongoose");  /\*\*   \* Conversation Schema   \*/  const ConversationSchema = new mongoose.Schema(    {      title: {        type: String,        maxlength: 64,        minlength: 1,        trim: true,        required: true,        unique: true      },    },    {      timestamps: { createdAt: "created\_at", updatedAt: "updated\_at" },    }  );  /\*\*   \* @typedef Conversation   \*/  module.exports = mongoose.model("Conversation", ConversationSchema); |

## **models\_\_Message.js**

|  |
| --- |
| const mongoose = require("mongoose");  /\*\*   \* Message Schema   \*/  const MessageSchema = new mongoose.Schema(    {      content: {        type: String,        required: true,      },      conversation: {        type: mongoose.Schema.Types.ObjectId,        ref: "Conversation",        required: true,      },      user: {        type: mongoose.Schema.Types.ObjectId,        ref: "User",        required: true,      },    },    {      timestamps: { createdAt: "created\_at", updatedAt: "updated\_at" },    }  );  MessageSchema.set("toObject", { virtuals: true });  MessageSchema.set("toJSON", { virtuals: true });  MessageSchema.virtual("timer").get(function () {    const date = new Date(this.created\_at).toLocaleTimeString("en-US");    return date;  });  /\*\*   \* @typedef Message   \*/  module.exports = mongoose.model("Message", MessageSchema); |

## **models\_\_User.js**

|  |
| --- |
| const mongoose = require("mongoose");  const bcrypt = require("bcryptjs");  const jwt = require("jsonwebtoken");  /\*\*   \* User Schema   \*/  const UserSchema = new mongoose.Schema(    {      email: {        type: String,        match: /^\S+@\S+\.\S+$/,        required: true,        unique: true,        trim: true,        index: true,        lowercase: true,      },      password: { type: String, required: true },    },    {      timestamps: { createdAt: "created\_at", updatedAt: "updated\_at" },    }  );  /\*\*   \* Hook   \*/  UserSchema.pre("save", async function (next) {    const user = this;    const hash = await bcrypt.hash(user.password, 10);    user.password = hash;    next();  });  /\*\*   \* Methods   \*/  UserSchema.methods.isValidPassword = async function (password) {    const user = this;    const compare = await bcrypt.compare(password, user.password);    return compare;  };  UserSchema.methods.token = async function () {    const user = this.toJSON();    const token = jwt.sign(user, process.env.JWT\_SECRET, {      expiresIn: process.env.JWT\_EXP,    });    return token;  };  /\*\*   \* @typedef User   \*/  module.exports = mongoose.model("User", UserSchema); |

## **index.js**

|  |
| --- |
| require("dotenv").config();  /\*\*   \* Server   \*/  const express = require("express");  const app = express();  /\*\*   \* Libs   \*/  const bodyParser = require("body-parser");  const jackrabbit = require("jackrabbit");  const mongoose = require("mongoose");  const passport = require("passport");  const strategies = require("./auth/passport");  const { authorize } = require("./auth/index");  const http = require("http").Server(app);  const io = require("socket.io")(http, {    path: "/websockets",    pingTimeout: 180000,    autoConnect: true,    pingInterval: 25000,    cors: {      origin: "\*",    },  });  /\*\*   \* ENV   \*/  const RABBITMQ\_URL = process.env.AMQP\_URL || "amqp://guest:guest@localhost";  const MONGODB\_URL = process.env.MONGODB\_URL || "mongodb://localhost/pad";  const PORT = process.env.PORT || 5000;  /\*\*   \* Models   \*/  const Conversation = require("./models/Conversation");  const User = require("./models/User");  const Message = require("./models/Message");  const Conversations = new Map();  const SocketsConversations = new Map();  /\*\*   \* Connect RabbitMQ   \*/  console.log("[RabbitMQ]: Connecting...");  const rabbit = jackrabbit(RABBITMQ\_URL);  const exchange = rabbit.default();  /\*\*   \* Connect MongoDB   \*/  console.log("[MongoDB]: Connecting...");  mongoose.connect(MONGODB\_URL, {    keepAlive: 1,    useNewUrlParser: true,    useUnifiedTopology: true,  });  /\*\*   \* Config server   \*/  app.set("port", PORT);  app.use(express.static(`${\_\_dirname}/app`));  app.use(bodyParser.json({ extended: true }));  app.use(bodyParser.urlencoded({ extended: true }));  app.use(passport.initialize());  passport.use("jwt", strategies.jwt);  /\*\*   \* Upsert conversation   \* @param {\*} title   \* @returns   \*/  const up\_conversation = async (title) => {    var conversation = await Conversation.findOne({ title });    if (!conversation) conversation = await new Conversation({ title }).save();    /\*\*     \* Make queue     \*/    const qname = `conversation@${conversation.\_id.toString()}`;    exchange.queue({      name: qname,      durable: true,    });    /\*\*     \* Consume / Subscribe to queue     \*/    const queue = exchange.queue({ name: qname, durable: true });    queue.consume(receiver);    return conversation;  };  /\*\*   \* Endpoints   \*/  app.get(    "/conversation/:title/messages",    authorize(),    async (req, res, next) => {      try {        const conversation = await up\_conversation(req.params.title);        const messages = await Message.find({ conversation })          .populate("user")          .sort({            created\_at: -1,          });        return res.status(200).json({ conversation, messages }).end();      } catch (error) {        return res.status(400).json({ error }).end();      }    }  );  app.post("/message", authorize(), async (req, res, next) => {    try {      const message = await new Message({        ...req.body,        user: req.user.\_id,      })        .save()        .then((doc) => doc.populate(["user", "conversation"]));      /\*\*       \* Publish message as JSON string       \*/      const qname = `conversation@${message.conversation.\_id.toString()}`;      exchange.publish(JSON.stringify(message.toJSON()), { key: qname });      return res.status(200).json(message).end();    } catch (error) {      return res.status(400).json({ error }).end();    }  });  app.post("/register", async (req, res, next) => {    try {      const user = await new User(req.body).save();      return res.status(200).json(user).end();    } catch (error) {      return res.status(400).json({ error }).end();    }  });  app.post("/login", async (req, res, next) => {    try {      const email = req.body?.email;      const password = req.body?.password;      const user = await User.findOne({ email });      var valid = await user.isValidPassword(password);      if (!valid) throw "Invalid.";      valid = await user.token();      return res.status(200).json({ jwt: valid, user }).end();    } catch (error) {      return res.status(400).json({ error }).end();    }  });  /\*\*   \* Socket   \*/  const \_socket = () => {    /\*\*     \* Handle socket connection     \*/    try {      io.use(async (socket, next) => {        const query = socket.handshake.query;        const conversation = query?.conversation;        const current = Conversations.get(conversation);        if (!current) Conversations.set(conversation, []);        SocketsConversations.set(socket.id, conversation);        return next();      }).on("connection", async (socket) => {        console.log(`[Socket]: Client connection ${socket.id}`);        /\*\*         \* Add connections         \*/        const conversation = SocketsConversations.get(socket.id);        const current = Conversations.get(conversation);        current.push(socket.id);        /\*\*         \* Socket diconnected remove user         \*/        socket.on("disconnect", async () => {          /\*\*           \* Clear connections           \*/          SocketsConversations.delete(socket.id);          var index = current.indexOf(socket.id);          if (index !== -1) current.splice(index, 1);          console.log(`[Socket]: Client disconnection ${socket.id}`);        });      });      console.log(`[Socket]: Successfully Connected`);    } catch (error) {      console.log(        `[Socket]: Unsuccessfully connected with error: ${error.message}`      );    }  };  /\*\*   \* Listen to RabbitMQ queue messages   \* @param {\*} data   \* @param {\*} ack   \*/  function receiver(data, ack) {    ack(); // Acknowledgement of RabbitMQ message    /\*\*     \* Convert json string to json object     \*/    const object = JSON.parse(data);    if (!object?.conversation?.title) return;    const current = Conversations.get(object?.conversation?.title);    if (!current) return;    current.forEach((e) => {      io.to(e).emit("message", object);    });  }  /\*\*   \* Run Server   \*/  \_socket();  http.listen(app.get("port"), () =>    console.log(`[Server]: Server started on port (${app.get("port")})`)  ); |

## **.env**

|  |
| --- |
| **AMQP\_URL='amqp://guest:guest@localhost'**  **RABBITMQ\_USER='guest'**  **RABBITMQ\_PASSWORD='guest'**  **PORT=5000**  **MONGODB\_URL=mongodb://localhost/pad**  **JWT\_SECRET=pad**  **JWT\_EXP='7d'** |

# **Rezultate**

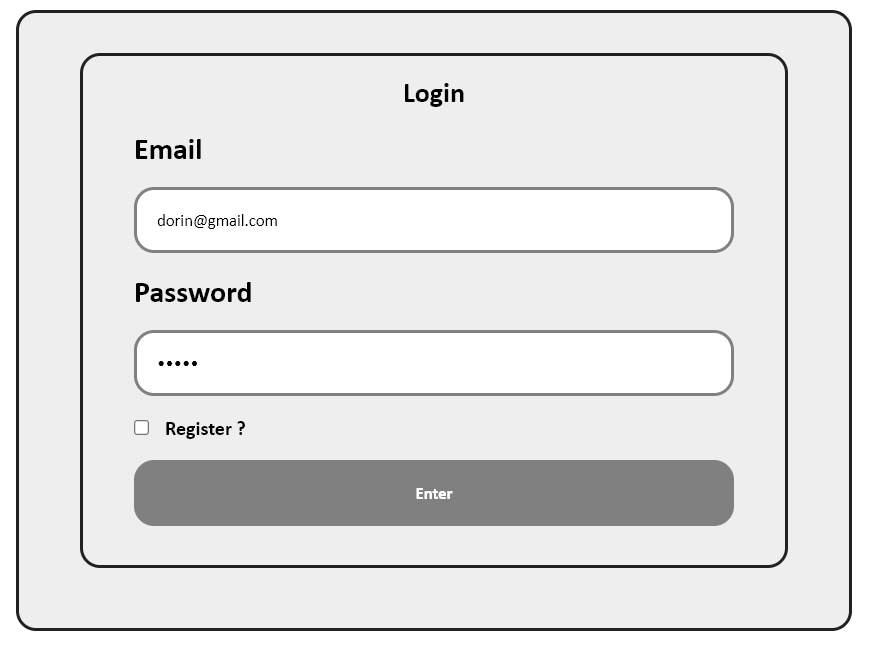


Figura 1 - Logarea

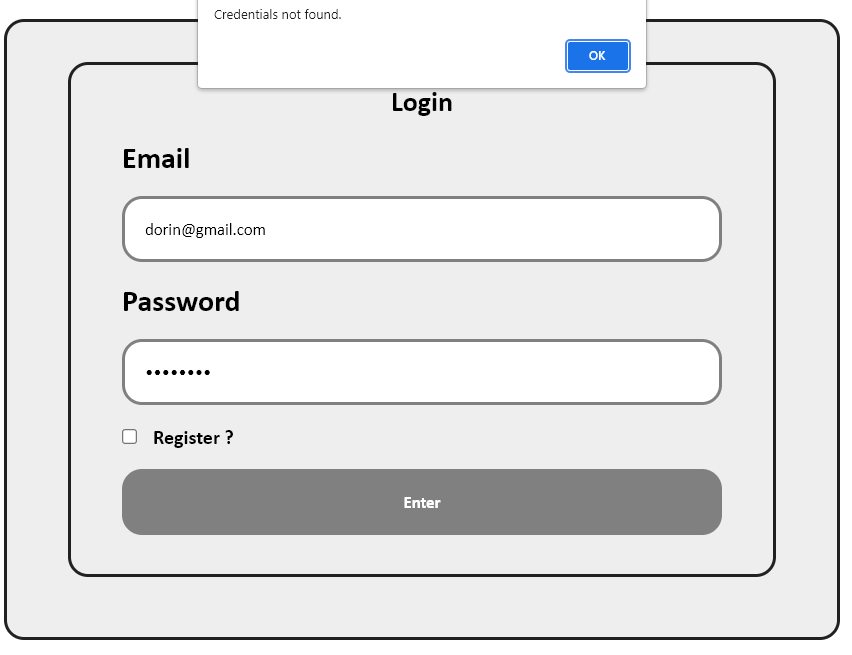


Figura 2 - Logare nereușită

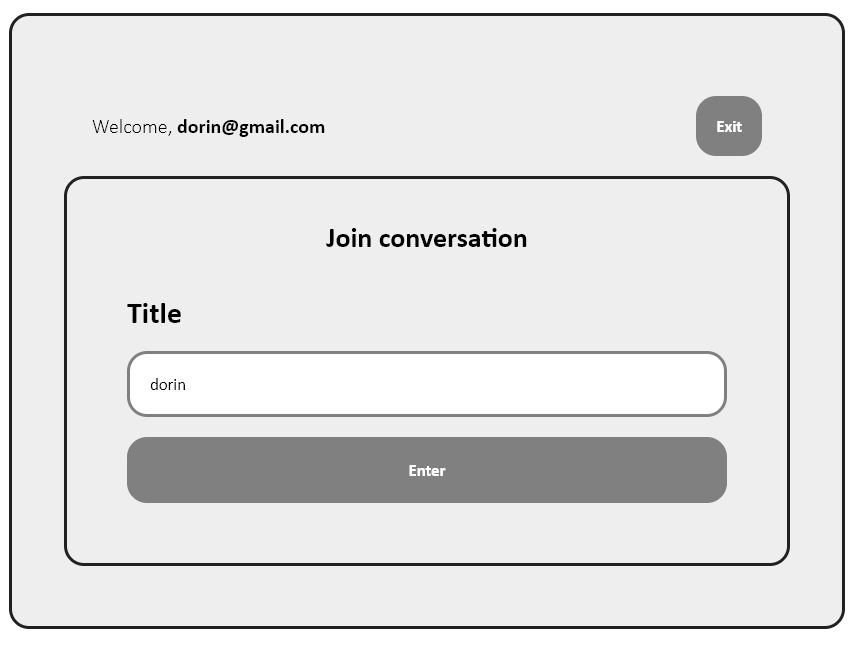


Figura 3 - Logare reușită

După logarea reușită trebuie să introducem numele conversației , în cazul cînd conversația nu există , conversația se va crea automat.

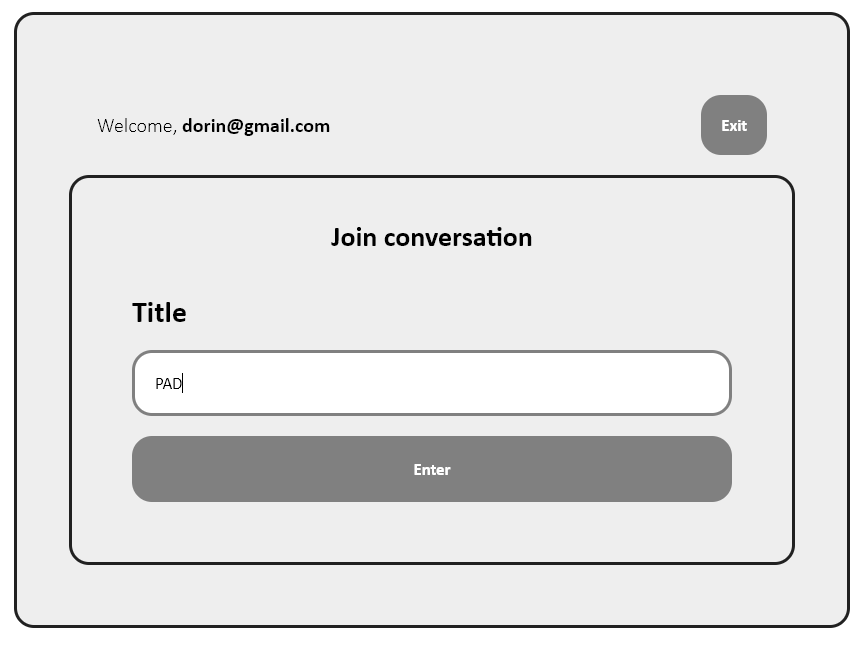


Figura 4 - Introducerea nume conversație

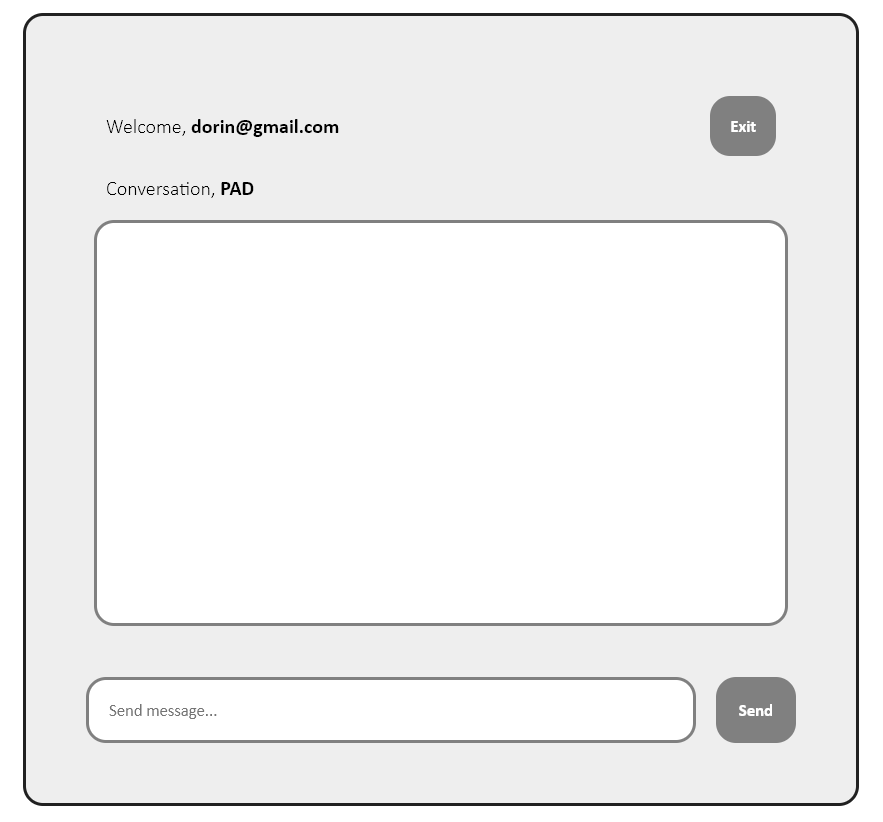


Figura 5 - Vizualizarea conversației

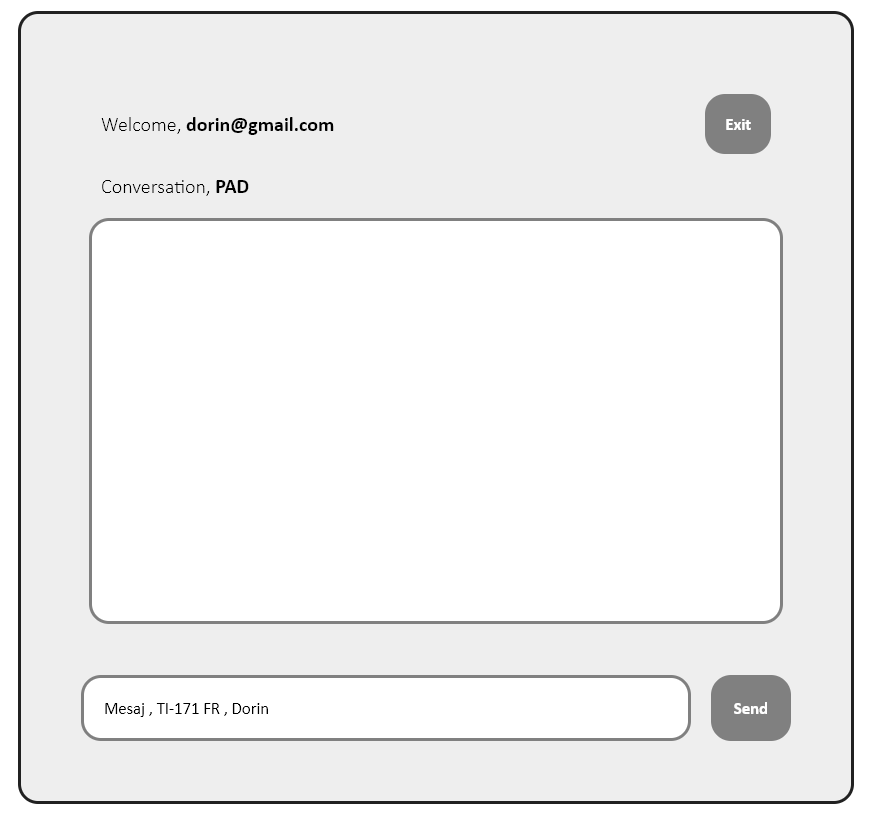


Figura 6 - Trimiterea mesajului



Figura 7 - Afișarea mesajelor

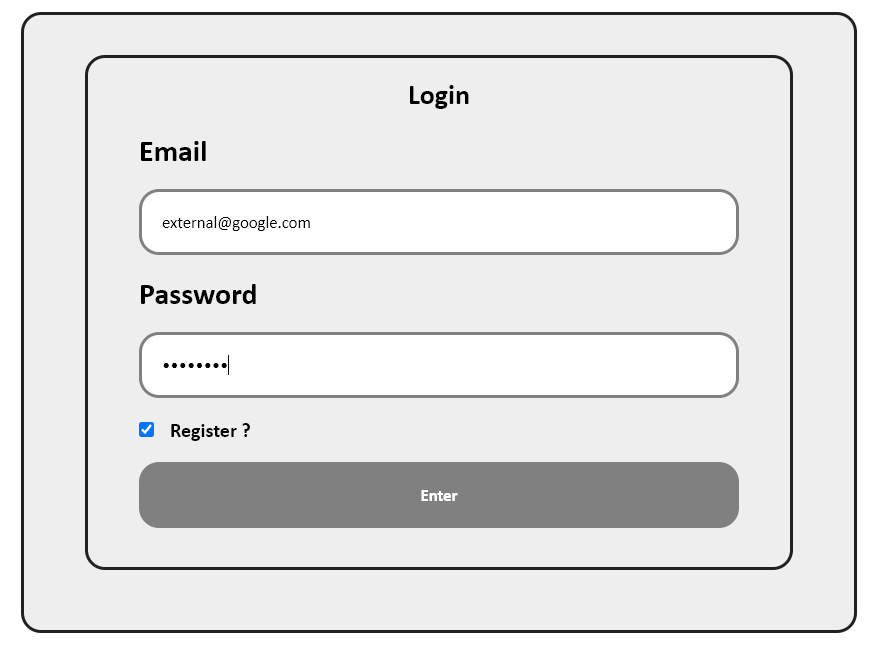


Figura 8 - Înregistrarea unui utilizator



Figura 9 - Accesarea conversației cu un utilizator extern

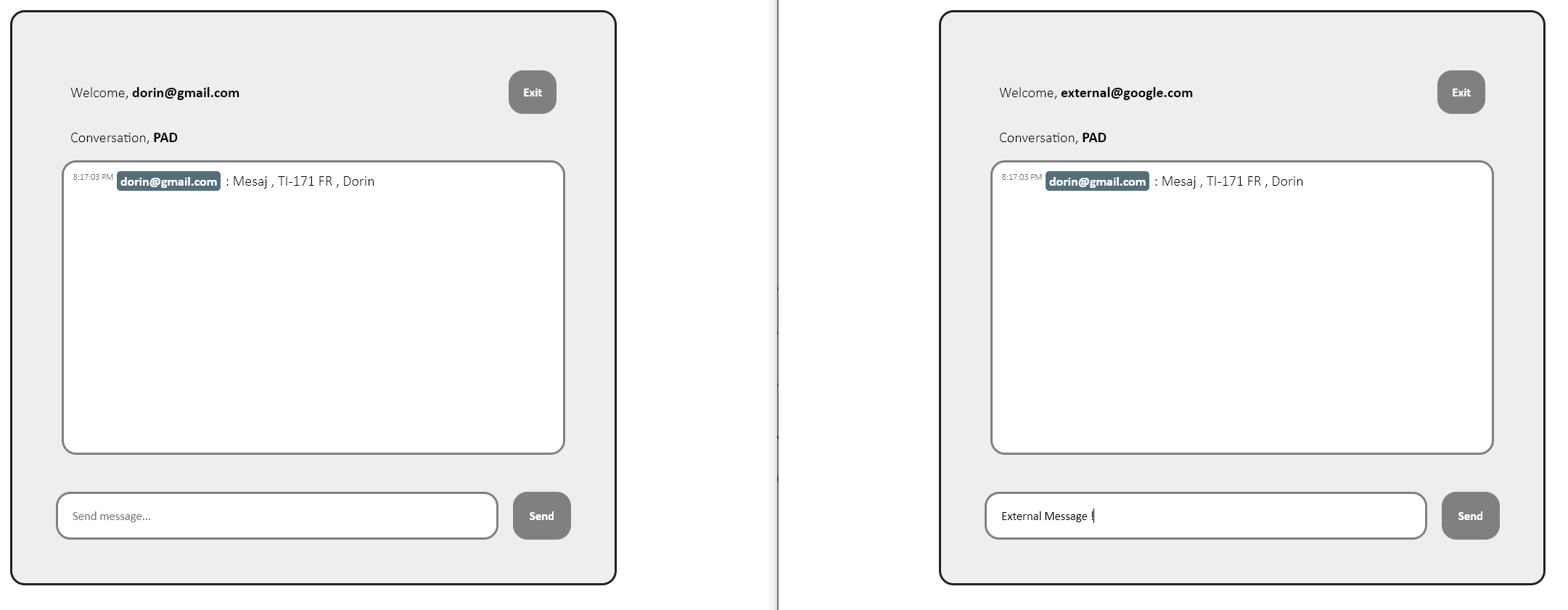


Figura 10 - Trimiterea mesajului în timp real

Primirea și transmiterea mesajelor în timp real se datorează implementării protocolului websocket.

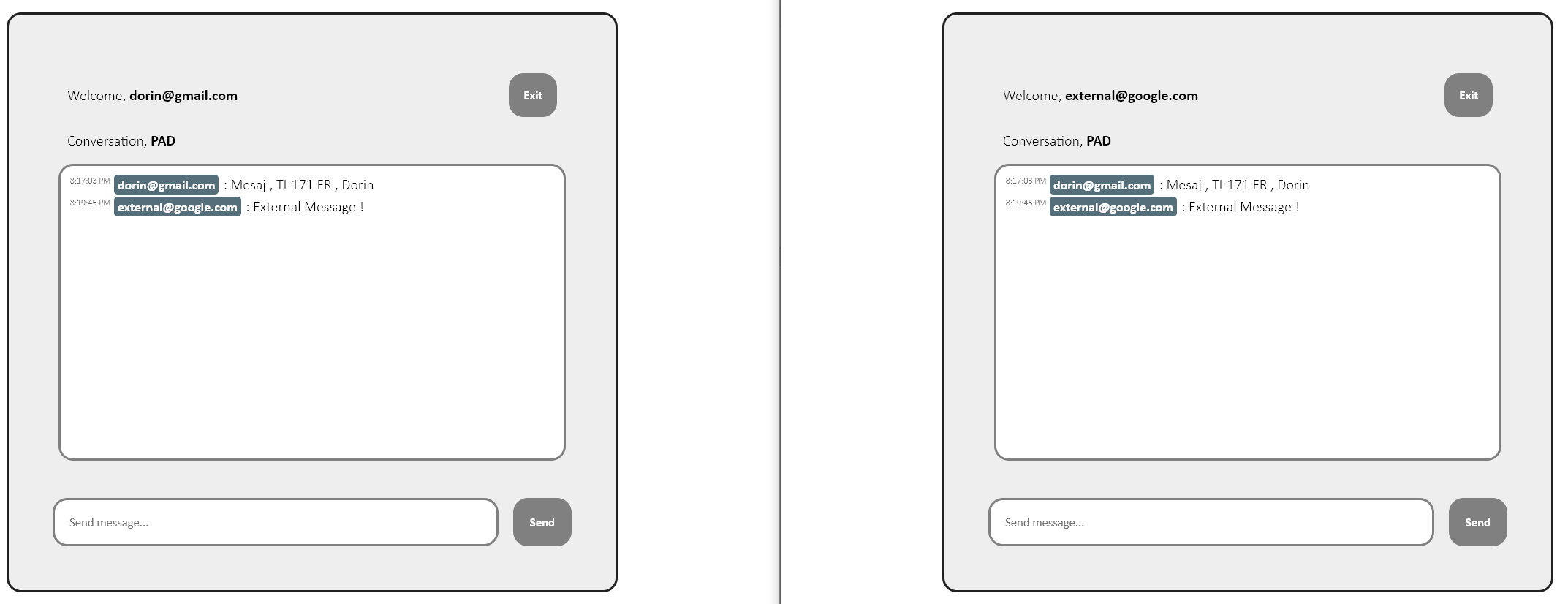


Figura 11 - Vizualizarea mesajelor în timp real

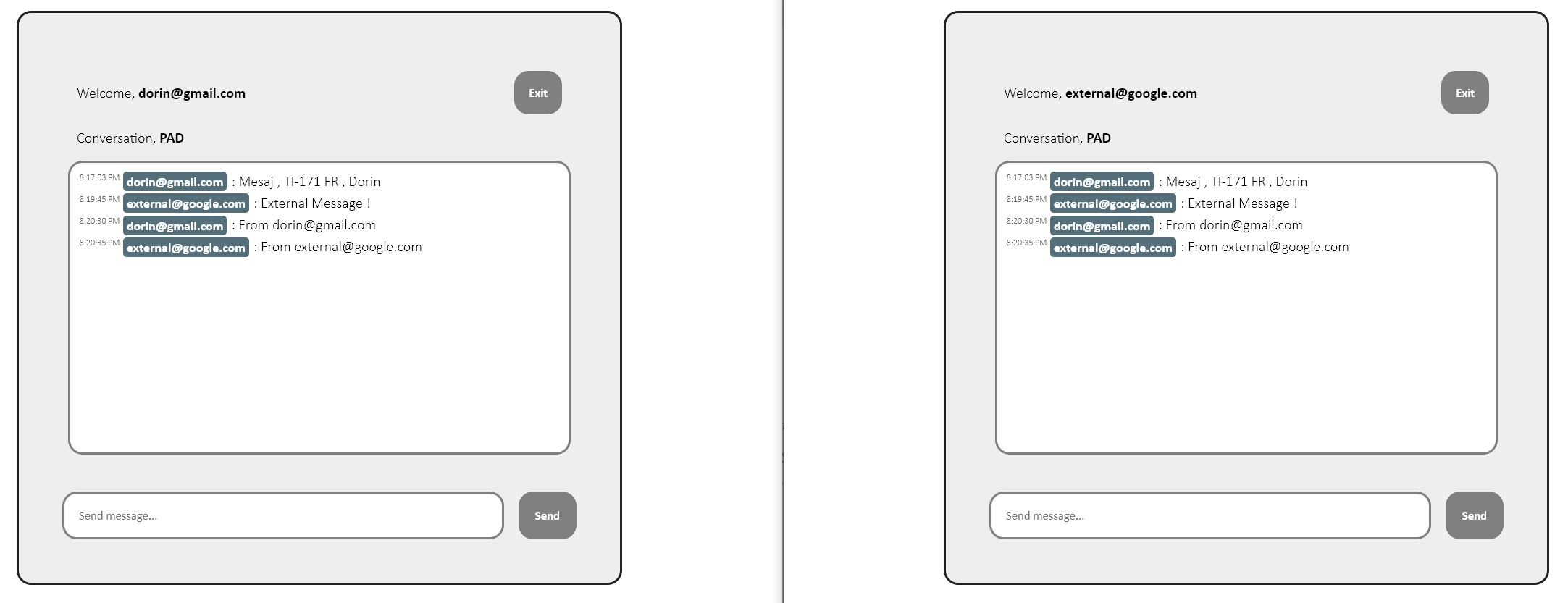


Figura 12 - Vizualizarea mai multor mesaje în timp real

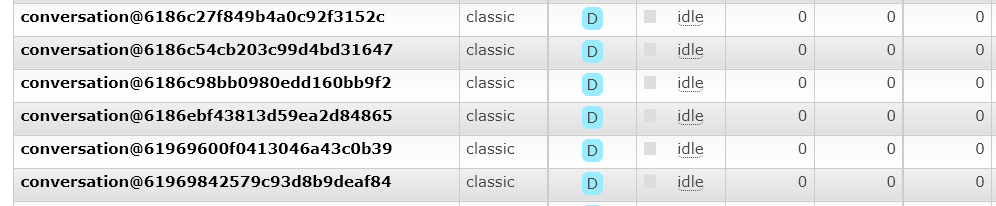


Figura 13 - Crearea de cozi în RabbitMQ

Pentru fiecare conversație se creează cîte o coadă (queue) în RabbitMQ , acest fapt ne permite să publicăm mesajele specifice conversației unice.

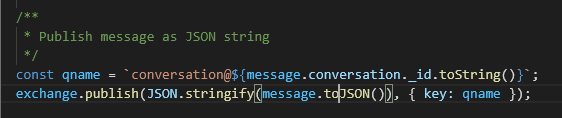


Figura 14 - Publicare mesajului în RabbitMQ

Mesajele sunt publicate ca un string JSON.

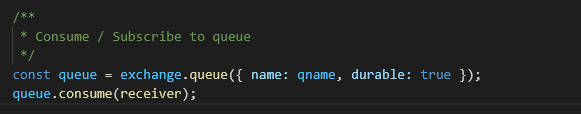


Figura 15 - Ascultarea mesajelor din RabbitMQ

Mesajele publicate sunt ascultate printr-un consumer.

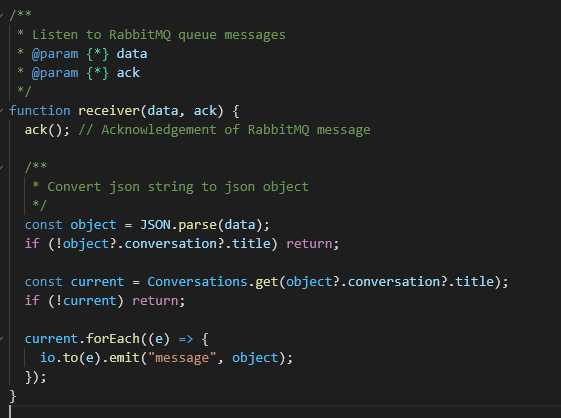


Figura 16 - Consumer

Consumer-ul parsează mesajul string JSON primit în format obiect JSON , dupa care în baza conținutului acestuia verifică clienții conectați și transmite mesajul prin socket.

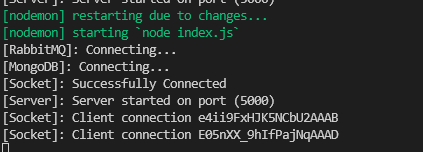


Figura 17 - Vizualizarea terminalului și clienților conectați

# **Concluzie**

În urma elaborării lucrării de laborator am căpătat cunoștințe de lucru cu protocolul websocket și broker-ul de mesaje RabbitMQ. Am implentat o logică de autentificare JWT pentru utilizatori folosind o baza de date MongoDB pentru salvarea conversațiilor , mesajelor și a utilizatorilor , ceea ce a facut posibil ca să elaborez o aplicație de mesagerie în care putem vizualiza mesajele precedente , înregistra și loga utilizatori , crea conversații unice în care putem comunica.