

**Fecha:** 09/07/2025

## Ejercicio 1: Chat Básico en Tiempo Real

Este ejercicio se enfoca en la comunicación en tiempo real y la persistencia de datos en un entorno asíncrono.

## Descripción del Ejercicio

Debes desarrollar el backend para una aplicación de chat muy simple. Los usuarios deben poder registrarse e iniciar sesión para obtener un token de autenticación. Una vez autenticados, podrán conectarse al chat a través de WebSockets para enviar y recibir mensajes en tiempo real. Todos los mensajes enviados deben ser guardados en una base de datos MongoDB.

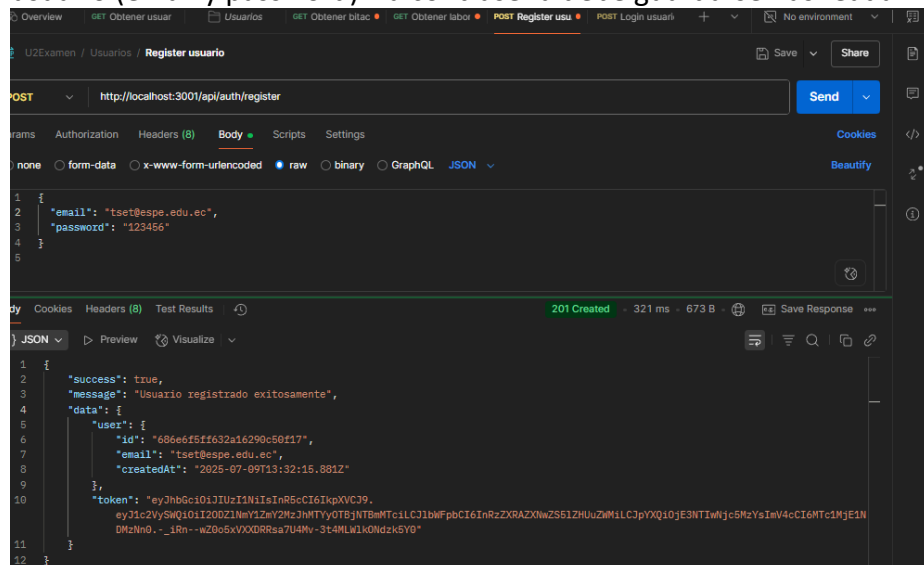
## Requisitos Técnicos

- **Backend:** Node.js con Express.
- **Base de Datos:** MongoDB con Mongoose.
- **Comunicación Real-Time:** WebSockets (puedes usar socket.io o ws).
- **Autenticación:** JWT (JSON Web Tokens).
- **Arquitectura:** Aplicar principios de Arquitectura Limpia (separación de capas).
- **Repositorio:** El proyecto final debe estar en un repositorio de GitHub.

## Requisitos Funcionales

## 1. Autenticación de Usuarios:

- Crear un endpoint POST /api/auth/register para registrar un nuevo usuario (email y password). La contraseña debe guardarse hasheada.



- Crear un endpoint POST `/api/auth/login` que valide las credenciales y devuelva un JWT.



- La conexión al WebSocket debe estar protegida. El cliente deberá enviar el JWT al momento de conectarse para ser validado. Si el token no es válido, la conexión debe ser rechazada.

```
const initializeWebSockets = (io) => {
  io.use(async (socket, next) => {
    try {
      const token = socket.handshake.auth.token;

      if (!token) {
        return next(new Error("Token de autenticación requerido"));
      }

      const decoded = verifyWebSocketToken(token);
      const user = await userRepository.findById(decoded.userId);

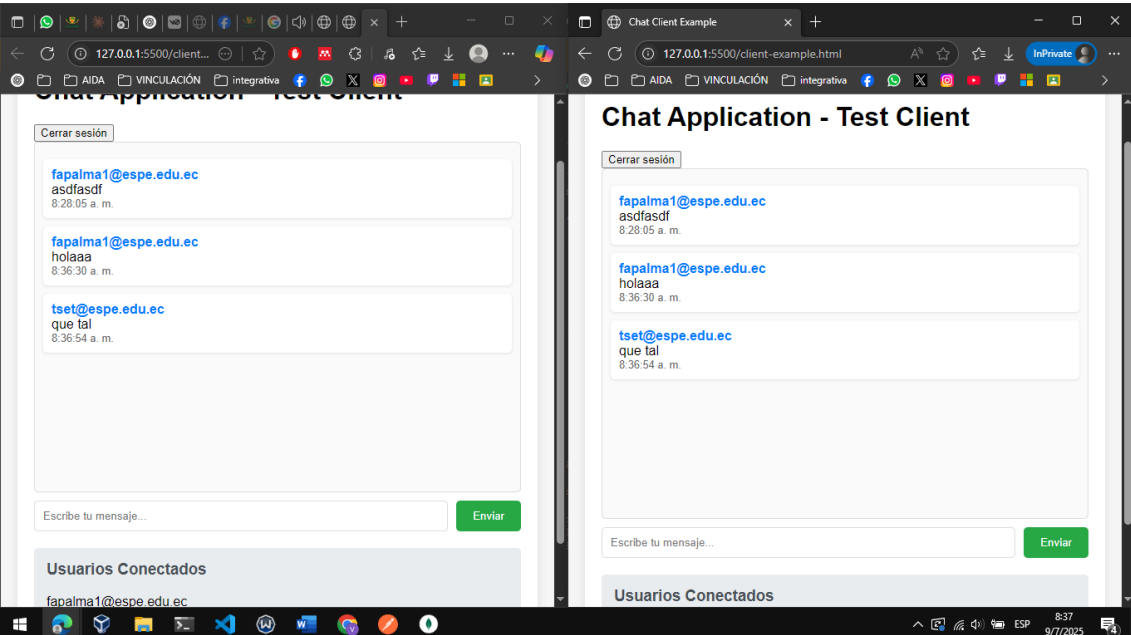
      if (!user) {
        return next(new Error("Usuario no encontrado"));
      }

      socket.userId = user._id.toString();
      socket.userEmail = user.email;
      next();
    } catch (error) {
      console.error("Error en autenticación WebSocket:", error.message);
      next(new Error("Token inválido"));
    }
  });

  io.on("connection", async (socket) => {
    console.log(`Usuario conectado: ${socket.userEmail} (${socket.id})`);
  });
}
```

- Cuando un cliente envía un evento `sendMessage` con un mensaje, el servidor debe:
  - Guardar el mensaje en la base de datos, asociándolo al usuario que lo envió.

- Transmitir (broadcast) ese mensaje a **todos** los clientes conectados.



The screenshot shows a web browser with two tabs. The active tab is titled 'Chat Client Example' and displays a chat application interface. The interface includes a 'Cerrar sesión' button, a list of messages from users 'fapalma1@espe.edu.ec' and 'tset@espe.edu.ec', and a 'Usuarios Conectados' section showing 'fapalma1@espe.edu.ec'. Below the chat interface is a table with three columns: 'ADD DATA', 'EXPORT DATA', and 'DELETE'. The table contains three rows of data, each representing a message object with fields like '\_id', 'text', 'user', 'createdAt', 'updatedAt', and '\_\_v'.

ADD DATA	EXPORT DATA	DELETE
<pre> _id: ObjectId('686e6e5f632a16290c50f0c') text: "asdfsdf" user: ObjectId('686e6ba0eb1ac4b505dcca76') createdAt: 2025-07-09T13:28:05.956+00:00 updatedAt: 2025-07-09T13:28:05.956+00:00 __v: 0 </pre>		
<pre> _id: ObjectId('686e705ef632a16290c50f1f') text: "holaaa" user: ObjectId('686e6ba0eb1ac4b505dcca76') createdAt: 2025-07-09T13:36:30.288+00:00 updatedAt: 2025-07-09T13:36:30.288+00:00 __v: 0 </pre>		
<pre> _id: ObjectId('686e7076f632a16290c50f27') text: "que tal" user: ObjectId('686e6f5ff632a16290c50f17') createdAt: 2025-07-09T13:36:54.944+00:00 updatedAt: 2025-07-09T13:36:54.944+00:00 __v: 0 </pre>		

### 3. Modelos de Datos (Mongoose):

- User: email, password.

```
const mongoose = require("mongoose");
const bcrypt = require("bcryptjs");

const userSchema = new mongoose.Schema(
  {
    email: {
      type: String,
      required: [true, "Email es requerido"],
      unique: true,
      lowercase: true,
      trim: true,
      match: [
        /^^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/,
        "Por favor ingresa un email válido",
      ],
    },
    password: {
      type: String,
      required: [true, "Password es requerido"],
      minlength: [6, "Password debe tener al menos 6 caracteres"],
    },
  },
  {
    timestamps: true,
  }
);
```

- **Message:** text, user (referencia al modelo User), createdAt.

```
const mongoose = require("mongoose");

const messageSchema = new mongoose.Schema(
  {
    text: {
      type: String,
      required: [true, "Texto del mensaje es requerido"],
      trim: true,
      maxlength: [1000, "Mensaje no puede exceder 1000 caracteres"],
    },
    user: {
      type: mongoose.Schema.Types.ObjectId,
      ref: "User",
      required: [true, "Usuario es requerido"],
    },
    createdAt: {
      type: Date,
      default: Date.now,
    },
  },
  {
    timestamps: true,
  }
);

// Índice para mejorar rendimiento en consultas por fecha
messageSchema.index({ createdAt: -1 });

module.exports = mongoose.model("Message", messageSchema);
```

### Estructura de Proyecto Sugerida (Arquitectura Limpia)

```
/src
|-- /api
|   |-- /routes      # Define las rutas de la API (auth.routes.js)
|   |-- /controllers # Controladores (req, res) (auth.controller.js)
```

```
|-- /domain
| |-- /models      # Esquemas de Mongoose (user.model.js, message.model.js)
| |-- /use-cases   # Lógica de negocio (register-user.use-case.js, save-message.use-case.js)
|-- /infrastructure
| |-- /repositories # Lógica de acceso a datos (user.repository.js)
| |-- /middlewares  # Middlewares de Express (auth.middleware.js)
| |-- /websockets   # Lógica de WebSockets (chat.handler.js)
|-- /config         # Configuración (db, variables de entorno)
|-- app.js          # Archivo principal del servidor Express
```

## Estructura del proyecto

