

Course : Introduction to Cryptography (ITC)

Session: July - Dec 2025

LAB ASSIGNMENT - 3

Deadline: 02-November-2025 (Sunday) Midnight

Objective

To design and implement a one-to-one private messaging application using Next.js and Socket.io, with messages persisted in a database for chat history and offline delivery.

Learning Outcomes

By completing this lab, students will be able to:

1. Build full-stack applications using Next.js with pages, API routes, and backend logic.
2. Implement one-to-one real-time messaging using WebSocket (Socket.io).
3. Store messages in a database (MongoDB) to maintain chat history.
4. Display previous messages when a user logs in or opens a chat.
5. Understand user session management and message routing for private communication.

Tools and Technologies

- Frontend & Backend: NodeJS, Next.js
- Real-time Communication: Socket.io
- Database: MongoDB (for storing messages and users)

Task Description

Part A – User Interface

1. **Login page:**
 - Users enter a **username** to connect.
 - Redirect to **chat page** after login.
2. **Chat page:**
 - Input field for **recipient username**.
 - Textbox for message input and **Send** button.
 - Chat window displaying **messages sent and received by this user only**, including **history**.
3. Style the interface with **CSS modules** or **Tailwind CSS** (optional)

Part B – Backend Setup (Next.js API + Socket.io)

1. Set up a **Next.js project** with API routes.
2. Integrate **Socket.io** for real-time messaging.
3. Maintain a **mapping of connected users**:

JavaScript

```
const users = {}; // { username: socket.id }
```

4. Handle **user registration on connection**:

JavaScript

```
socket.on("register_user", (username) => {  
  
  users[username] = socket.id;  
  
});
```

5. Handle **sending private messages with storage**:

JavaScript

```
socket.on("send_message", async ({ sender, receiver, text }) => {  
  
  // Store message in MongoDB  
  
  await Message.create({ sender, receiver, text, timestamp: new  
Date() });  
  
  // Send message to the receiver if online  
  
  const receiverSocketId = users[receiver];  
  
  if (receiverSocketId) {  
  
    io.to(receiverSocketId).emit("receive_message", { sender,  
text });  
  
  }  
  
});
```

Part C – Frontend Real-time Communication

1. Establish **WebSocket connection** on the chat page.
2. When a message is sent:
 - Emit it to the server with **sender**, **receiver**, and **text**.
 - Optionally display it locally in sender's chat window.
3. Listen for **receive_message** events from the server and **update chat window dynamically**.
4. On **page load**, fetch chat history from the backend API and display previous messages:

JavaScript

```
const res = await
fetch(`/api/messages?user1=${username}&user2=${recipient}`);

const history = await res.json();

setMessages(history);
```

Part D – Optional Enhancements

1. Display **“User is online/offline” status**.
2. Show **“User is typing...” notifications**.
3. Enable **editing or deleting sent messages**.

Submission Guidelines

- Upload the full project on **GitHub**.
- Include a **README.md** explaining:
 - Project overview
 - Steps to install and run the app locally
 - Screenshots of chat interface
 - Architecture
 - Technologies used
 - Learning outcomes

END