

Secure Chad

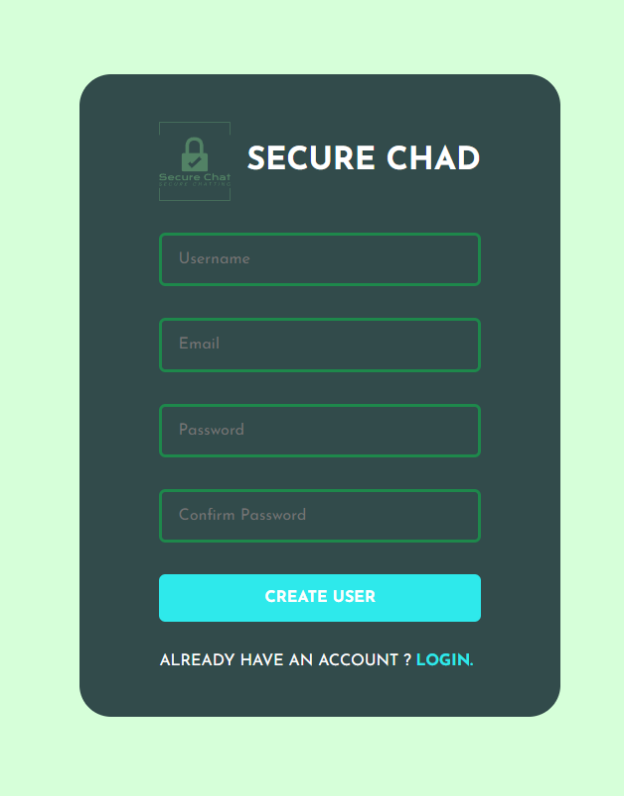
Our project is a full stack MERN project which means it utilizes MongoDB Express React and Node Js. Our front-end is written in React and in the Backend we have utilized express to run on localhost:3000 and in localhost we are collecting data which we integrated with our code via mongoose package. This package gets the Users name, emails, avatar, message sender, message receiver, messages itself(which are encrypted) and automatically stores them into two categories: chat and users tables respectively.

In addition , we decided to use the “iterative waterfall model”. From our experiences in older projects, the iterative model provides us an application with accurate and less buggy software if the process is employed according to schedule.

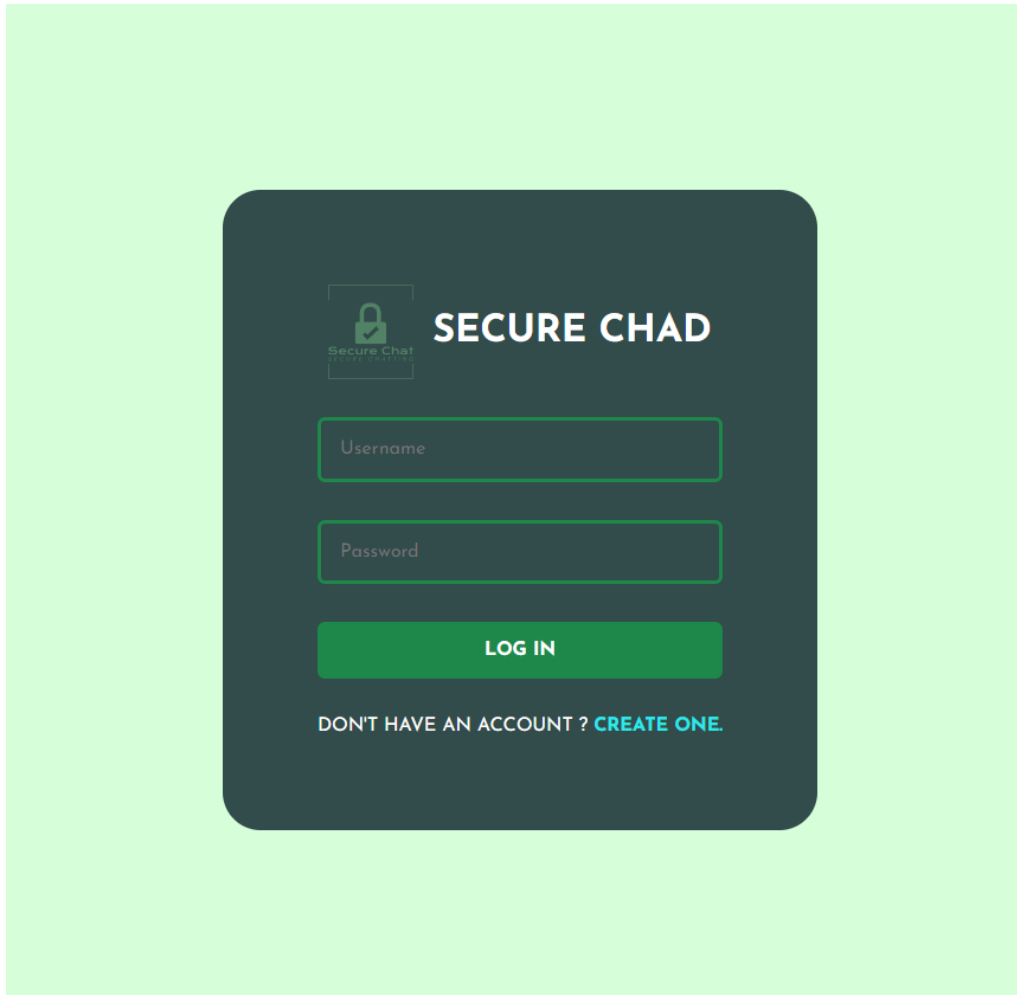
Unfortunately, since we repeat each process one by one to find out all defects and errors, this can cause time-wasting problems as a result. Those are only problems we have encountered so far. Hence we thought it would be beneficial for small team projects like ours.

Here is our brief summary of our project:

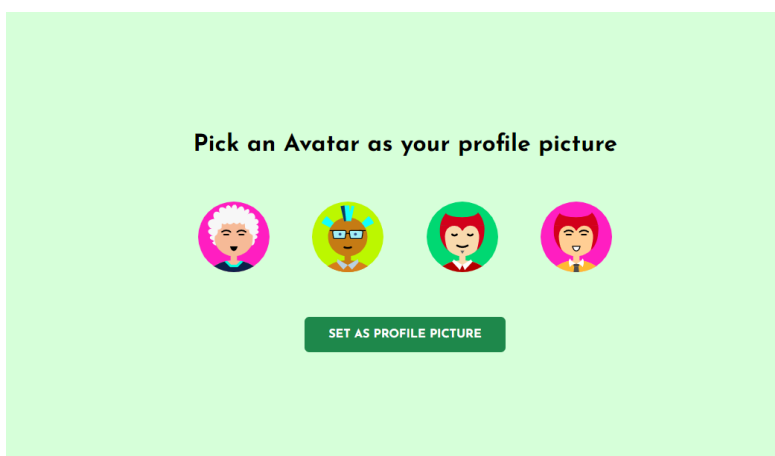
We have our first screen which is our registration page.

The image shows a registration form for 'Secure Chad' centered on a light green background. The form itself is a dark teal rounded rectangle. At the top left of the form is a small logo with a padlock icon and the text 'Secure Chat'. To the right of the logo, the title 'SECURE CHAD' is displayed in white, bold, uppercase letters. Below the title, there are four input fields, each with a light green border and placeholder text: 'Username', 'Email', 'Password', and 'Confirm Password'. Under these fields is a bright green button with the text 'CREATE USER' in white, uppercase letters. At the bottom of the form, there is a line of text: 'ALREADY HAVE AN ACCOUNT ?' followed by the word 'LOGIN.' in blue, uppercase letters, where 'LOGIN.' is a hyperlink.

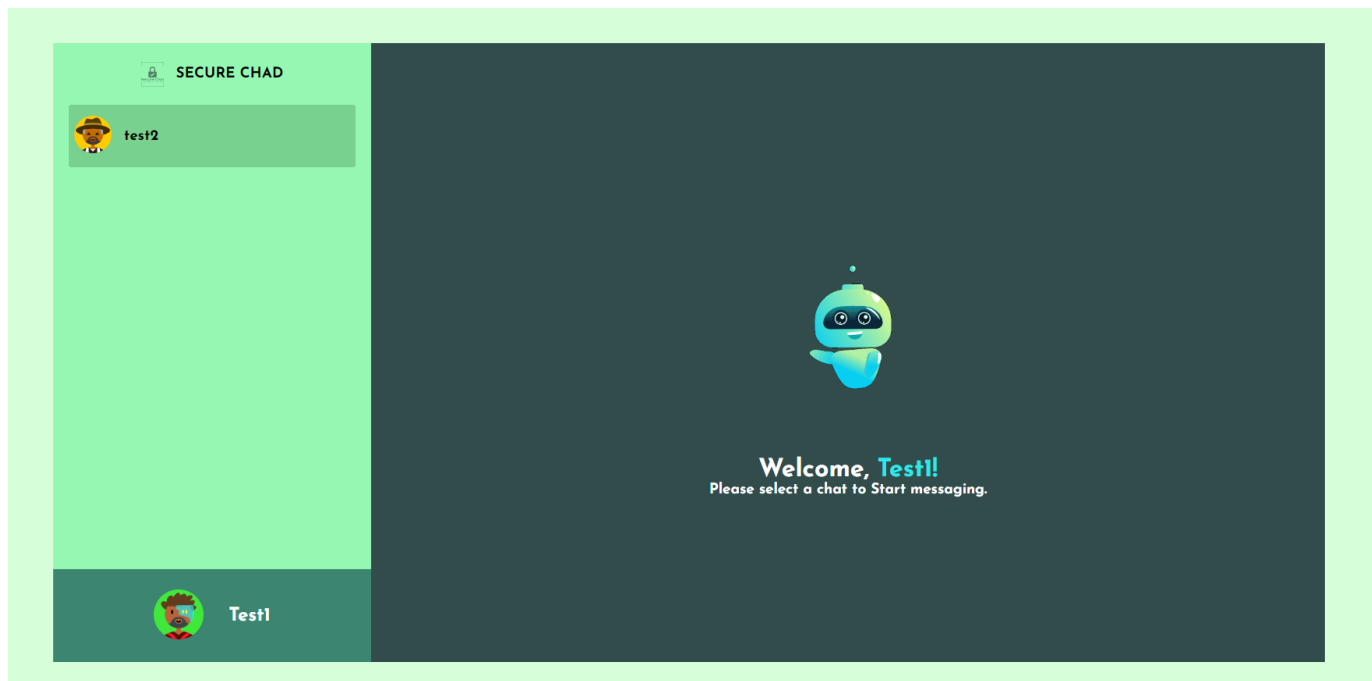
or alternatively if user is already registered we have a login page:



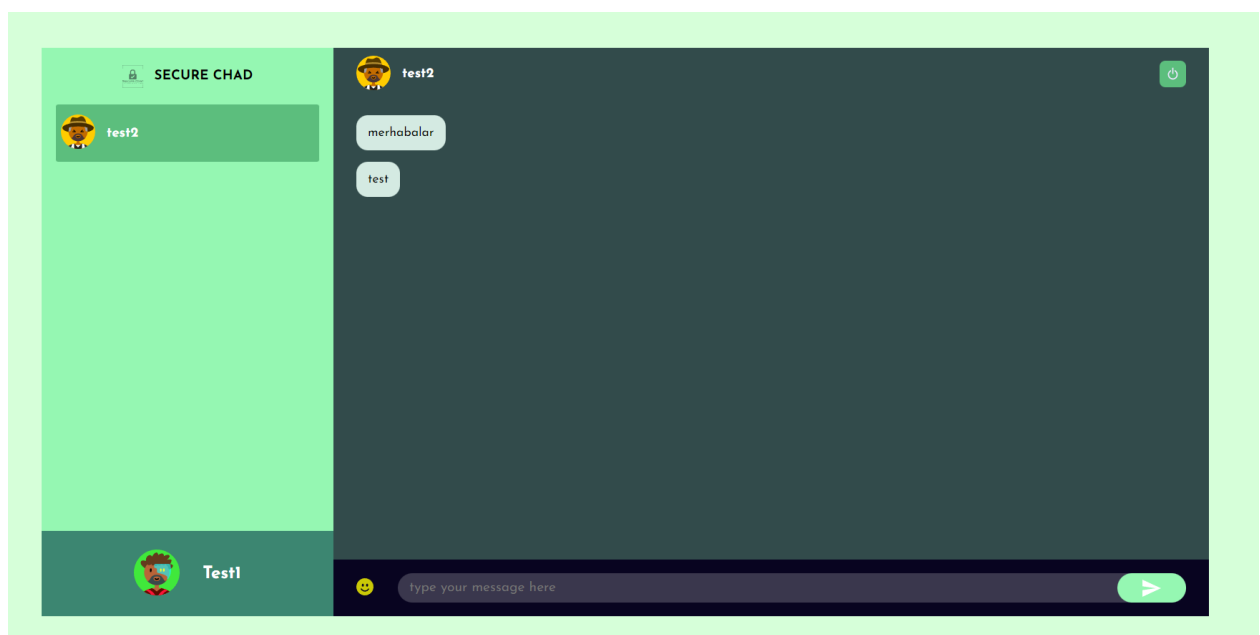
If a user has created a fresh account after registering they get to choose an avatar for themselves.



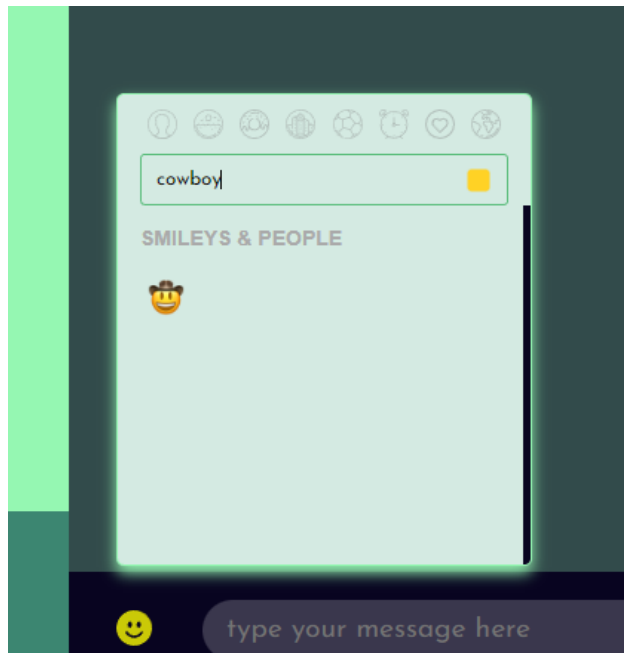
After setting up their account the user is greeted with a user screen that is located on their left screen as you can see.



And then you can pick a conversation up with the user and proceed to chat up. while chatting our AES encryption is doing its work: encrypting and decrypting respectively the encrypted messages are saved to mongodb with mongoose package.for further conversations of offline chat or logging the chat.



For chatting we have also included a emoji picker and implemented a search bar for emoji searching we will try to implement this feature to our users sidebar because a concern of ours for future development is after some usage of the app the users tab can be clustered and full we can try to solve this issue by searching the chats.



chat.messages

Documents Aggregations Schema Explain Plan Indexes Validation

FILTER { field: 'value' }

ADD DATA

VIEW

```
  _id: ObjectId('63578198089aef66a38fc20d')
  message: Object
    text: "merhabalar"
  users: Array
    0: "6357817a089aef66a38fc206"
    1: "6357814a089aef66a38fc200"
  sender: ObjectId('6357817a089aef66a38fc206')
  createdAt: 2022-10-25T06:26:32.422+00:00
  updatedAt: 2022-10-25T06:26:32.422+00:00
  __v: 0
```

```
  _id: ObjectId('63578199089aef66a38fc20f')
  message: Object
    text: "test"
  users: Array
    0: "6357817a089aef66a38fc206"
    1: "6357814a089aef66a38fc200"
  sender: ObjectId('6357817a089aef66a38fc206')
  createdAt: 2022-10-25T06:26:33.761+00:00
  updatedAt: 2022-10-25T06:26:33.761+00:00
  __v: 0
```

In the table above as you can see the messages are logged with their sender object id, sender, created message's date & time. And in the table below as you can see we have users table which holds information about the users

chat.users 3 DOCUMENTS INE

Documents Aggregations Schema Explain Plan Indexes Validation

Displaying documents 1 - 3 of 3

```
{
  "_id": ObjectId('6344350b3d684ee58d9be7ea'),
  "username": "Uhmada",
  "email": "random@gmail.com",
  "password": "$2b$10$QaJDE3jeEVxavud46cr4tuJewPAPf7K8DQphee0Xf17UuBVU6B4Ku",
  "isAvatarImageSet": true,
  "avatarImage": "PHN2ZyB4bWxucz0iaHR0cDovL3d3dy53My5vcmcvMjAwMC9zdmciIHZpZXh0b3g9IjAgMC..."
  "__v": 0
}
```

```
{
  "_id": ObjectId('6344354d3d684ee58d9be7f1'),
  "username": "Uhmada2",
  "email": "random2@gmail.com",
  "password": "$2b$10$9yS.OWOkCC.23kzoERgBZeulKrjueU6h/z7ABZN0jHBAbTdwHY7oC",
  "isAvatarImageSet": true,
  "avatarImage": "PHN2ZyB4bWxucz0iaHR0cDovL3d3dy53My5vcmcvMjAwMC9zdmciIHZpZXh0b3g9IjAgMC..."
  "__v": 0
}
```

```
{
  "_id": ObjectId('634719253d091a0378ac6257'),
  "username": "Uhmada3",
  "email": "test3@gmail.com",
  "password": "$2b$10$msiUtahBJFwXr0XrzCZ12Ot8vsWjmpDzEVD06/.eWbJI0cq/hr4D6",
  "isAvatarImageSet": true,
  "avatarImage": "PHN2ZyB4bWxucz0iaHR0cDovL3d3dy53My5vcmcvMjAwMC9zdmciIHZpZXh0b3g9IjAgMC..."
  "__v": 0
}
```

Here are the coding parts: this part is our model for the mongoose package.

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

const MessageSchema = mongoose.Schema(
  {
    message: {
      text: { type: String, required: true },
    },
    users: Array,
    sender: {
      type: mongoose.Schema.Types.ObjectId,
      ref: "User",
      required: true,
    },
  },
  {
    timestamps: true,
  }
);

module.exports = mongoose.model("Messages", MessageSchema);
```

And this is our controller part of the code: this particular code segment gets the messages.

```
const Messages = require("../models/messageModel");

module.exports.getMessages = async (req, res, next) => {
  try {
    const { from, to } = req.body;

    const messages = await Messages.find({
      users: {
        $all: [from, to],
      },
    }).sort({ updatedAt: 1 });

    const projectedMessages = messages.map((msg) => {
      return {
        fromSelf: msg.sender.toString() === from,
        message: msg.message.text,
      };
    });
    res.json(projectedMessages);
  } catch (ex) {
    next(ex);
  }
};
```

And this particular part of the code segment adds to the database:

```
module.exports.addMessage = async (req, res, next) => {
  try {
    const { from, to, message } = req.body;
    const data = await Messages.create({
      message: { text: message },
      users: [from, to],
      sender: from,
    });

    if (data) return res.json({ msg: "Message added successfully." });
    else return res.json({ msg: "Failed to add message to the database" });
  } catch (ex) {
    next(ex);
  }
};
```

We also have a user model coding part as well.

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

const userSchema = new mongoose.Schema({
  username: {
    type: String,
    required: true,
    min: 3,
    max: 20,
    unique: true,
  },
  email: {
    type: String,
    required: true,
    unique: true,
    max: 50,
  },
  password: {
    type: String,
    required: true,
    min: 8,
  },
  isAvatarImageSet: {
    type: Boolean,
    default: false,
  },
});
```

```
    password: {
      type: String,
      required: true,
      min: 8,
    },
    isAvatarImageSet: {
      type: Boolean,
      default: false,
    },
    avatarImage: {
      type: String,
      default: "",
    },
  });

module.exports = mongoose.model("Users", userSchema);
```

How does the System Work:

First of all, our npm packages need to be downloaded to run the program. We explained this very clearly in the readme section of the project. In two terminals, we are changing the directory to the “public” and “server” folders separately. After we reach the location of packages, we basically write the “npm i” command for both of them to install necessary packages that are written in the “packages.json” file. Then we run the “yarn start” command for both terminals.

Congrats, now we can reach the application that we need. This process opens the server system and client side socket protocol. Socket is the name of the protocol that gives us instantaneous data transfer.

```
const handleSendMsg = async (msg) => {
  const data = await JSON.parse(
    localStorage.getItem(process.env.REACT_APP_LOCALHOST_KEY)
  );
  console.log("Mesaj:" + msg);
  let key = KeyExpansion(genRanHex(32));
  let encrypted = '';
  let hex = hexconvert(msg);
  console.log("Hex:" + hex);
  let loop = parseInt(hex.length / 32);
  for (let x = 0; x < loop; x++) {
    encrypted += AES_128_Encryption(hex.substring(32 * x, 32 * (x+1)), key) /// encryption
  }
  console.log(encrypted);
  socket.current.emit("send-msg", { // sokete giden mesaj
    to: currentChat_id,
    from: data_id,
    encrypted,
    key,
  });
  await axios.post(sendMessageRoute, {
    from: data_id,
    to: currentChat_id,
    message: msg,
  });
}
```

After seeing the above code block, it is quite simple to understand how the code works outside the background. The **red** part represents the backend part that takes the message input from the user and brings it to our code. The **green** part is the encryption part of the message. The **blue** part represents how the socket works. We're publishing our encrypted message, our randomly generated key, chat_id and data_id to the room. The last two parameters are necessary for the database. Finally the **orange** part of code is all about collecting the detailed message information in our database. Of course this part works with the bottom part together.

```
await axios.post(sendMessageRoute, {
  from: data_id,
  to: currentChat_id,
  message: msg,
});

const msgs = [...messages];
msgs.push({ fromSelf: true, message: msg });
setMessages(msgs);
};
```


Until this time, we can send the encrypted message to our system and database, but how can we get the message from a stranger. After we send a message, we change the current version of socket as we send new data.

```
useEffect(() => {
  if (socket.current) {
    socket.current.on("msg-recieve", (encrypted,key) => { // serverdan gelen mesaj yeri
      let decrypted = ''
      for (let i = 0; encrypted.length / 32 !== i; i++) {
        decrypted += AES_128_Decryption(encrypted.substring(32*i,32*(i+1)),key)
      }
      console.log("hex şifresiz: " + decrypted);
      let original = ''
      for (let i = 0; parseInt(decrypted.length / 4) > i; i++) {
        if(decrypted.substring((4 * i),4 * (i+1)) === '0000') {
          break;
        }
        console.log(decrypted.substring((4 * i),4 * (i+1)))
        original += String.fromCharCode(
          parseInt(decrypted.substring((4 * i),4 * (i+1)),16)
        )
      }
      console.log("Server'dan gelen mesaj (şifresiz):" + original);
      setArrivalMessage({ fromSelf: false, message: original });
    });
  }
}, [1]);
```

When we get a new message from our socket, first the system tries to decrypt it. After the decryption part, write it to our screen.

This is the most basic approach to our secure chat system. If we want to put this project in real world, we need some funds for;

- Deploy the server part (backend) of our project to the AWS like server base.
- Deploy the client part (frontend) of our project to a domain.

Making a guess of this process's cost is so unpredictable, because it depends on the user count of our system.

If I need to give some hints about what is the next feature of our project;

- We need to implement group chat rooms like whatsapp.
- We will find a better way to hide the process of our message key.
- We will add the user's name on his/her message's above.
- We will add a search bar among friends. Maybe friend requests.

For now, it's too hard to make some precise predictions about our project at the moment. But we aspire to develop the project with common software development standards and sustainably.

About the Security Part:

For our message encryption. We are using handwritten AES-128 bit encryption since it's one of the most popular and used encryption methods in which supports 16 bit characters being encrypted into cipher text. But for now we still have a lack of security measures in server side and client side.

Apart from the encryption algorithm working as expected, the key we are using for encryption is still in early stages. In order to test the algorithm, we made a temporary key generator. After encryption we send it to another client('s) along with cipher text to decrypt. This of course creates a security breach. We are planning some solutions to resolve this issue. For example, we may create a key distributor service on the server side, giving clients a temporary key to encrypt and decrypt with a small TTL (Time to live) value.

Another issue is storing messages in databases. In order to display offline pages we are storing messages in Our mongodb database as mentioned above as plain (not encrypted) text. Our plan for now will be storing all necessary data as encrypted via custom cipher functions from node.js packages.

```
1900005528-Ahmet Kaan Memioğlu  
1900005485-Emreçan Üzüm  
1900003587-Şükrü Erim Sinal
```