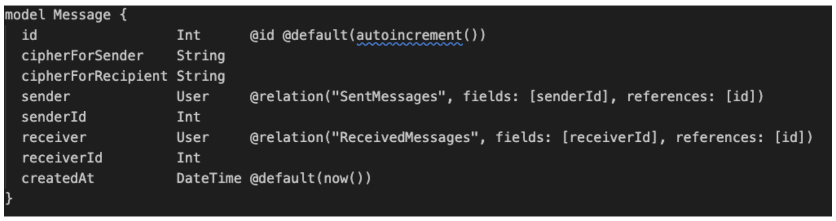
CIS-446 Term Project

E2E Simple encrypted chat app

By Joseph Cornell

E2E chat app system overview

For my term project I chose to implement a secure, realtime chat application built with nextjs15, prisma ORM, and sqlite3 for database functionality, with appropriate apis for any data transfer or any other needs of the system and a focus on user privacy and data protection. The system is designed to allow users to register with their credentials and receive their private key that they should save or store away for later, upon login the user will provide their credentials and their private key, if successful the user will be logged in and redirected to their login page where they are able to search, add and delete friends, view their current friends and conversations, or click on any profile to start a conversation with that user. These messages within the chat are encrypted, so even if the database or server for our application became compromised at any point, the data would not have been in a viewable state, the model within the database stores the encrypted content for the user *and* the receiver. The recipient's public key is used to encrypt the content for the other user and the current user's public key is used to encrypt the content for the current user. When the chat page loads, messages from the



```
model Message {
  id                 Int      @id @default(autoincrement())
  cipherForSender    String
  cipherForRecipient String
  sender             User     @relation("SentMessages", fields: [senderId], references: [id])
  senderId           Int
  receiver           User     @relation("ReceivedMessages", fields: [receiverId], references: [id])
  receiverId         Int
  createdAt          DateTime @default(now())
}
```

other person and the current user are decrypted using the current user's private key.

Messages in this application are also handled using web sockets to allow real time communication and modification of the conversations. Users can perform CRUD commands on their messages, allowing them to view, edit and delete them after sending in case a mistake was made. Other parts of the application don't require such tolerances so I decided it was fine to keep friend requests, adding and removal, simple. I performed the updates for this data by polling the database around every 3 seconds. Caveats to this system's design is the storage of private keys/passcode pairs for each user. If at any point a user loses their private key or forgets their passcode, their data is lost.

There are plenty of potential threat vectors that could be applied to this application, since this was a simple application I left other parts besides messages (such as a user's friends list data) of the datastream unencrypted. This would allow for unauthorized access/viewing of the users information and potentially even more ill side effects not realized in this simple E2E chat app.

Another consideration is unauthorized access of the users device/account if they are already logged in. This outside agent could perform any unauthorized action in the system/ impersonate/phish and the original user would not have very many ways to resolve this

Overall this application would do much better with another iteration of development to polish up details but definitely fulfills the requirements of end to end encryption of chat data and was a great learning opportunity!