For implementation of the database I chose RoomDB, which is an high level Kotlin library that provides abstraction for creating and managing SQlite databases. RoomDB's can be implemented through several layers: the tables, data access object, repository, and the AppViewModel [**?**]. The tables are defined using special syntax that correlates to sqlite table qualities like Primary keys and columns. Tables are defined using kotlin data classes using syntax like the following.

Listing 1: Example of table definition.

```
@Entity(tableName = "contacts_table")
data class Contacts (
    @PrimaryKey(autoGenerate = false)
    val user: String,
    val key: String,
    val address: String
    )
```

Four tables are need for the basic functions of this application: the contact table, primary user table, conversation table, and the message table.

- The contact table has three columns: Contact name(Primary key), key, and last updated address
- Primary user table has two columns: Id(primary key), hidden service address. Only one entry should ever been in this table.
- The conversation table has three columns: ID(Primary key), the contact name, and the conversation name
- The message table has five columns: ID(Primary key), the contact name, conversation name, message, and time stamp

Data Access Objects(DAO) are used to define query's, insertions, deletions, and other SQL activities. Due to the nature of IO like database access it is best to use coroutines to preform database access independently of the Current thread. Coroutines are simple threads that allow for non-blocking code to be easily implemented [**?**]. This prevents the blocking of UI or of the message server when accessing the database. For the purpose of this application four insertion and four queries are necessary of the base functionality of the app.

Listing 2: Example of DAO definition.

```
@Dao
interface DAO {
    // Adds a contact to the contacts table, takes the contact data class from
    // the Table definition as input and returns nothing.
    // If there is a conflict like a non-unique
    // primary key the function will call a fail exception
    @Insert(onConflict = OnConflictStrategy.ABORT)
    suspend fun addContact(user: Contacts)

    // Adds a primary user to the primary user table.
    // This is where data like username and current hidden address are stored.
    // On conflict the insertion fails and thows a fail exception
    @Insert(onConflict = OnConflictStrategy.ABORT)
    suspend fun addPrimaryUser(user: PrimaryUser)

    // Adds a contact to a coversation by
    // making an entry in the conversations table.
    // This is how contacts are grouped into conversations.
    // Setup in a way to allow for group
    // conversations in the future but currently this is
    // missing feature.
    @Insert(onConflict = OnConflictStrategy.ABORT)
    suspend fun addToConversation(convo: Conversations)

    // Inserts a message into the message database.
    // Messages from all users are
    // stored here even messages from the primary user.
    // They are associated to conversations by the convo ID.
    @Insert(onConflict = OnConflictStrategy.ABORT)
    suspend fun addMsg(msg: Msgs)

    // Selects all msgs that match the conversation ID
    // and returns them ordered by their timestamp.
    // This is used in the UI to
    // show the message history of a coverstation.
    @Query("Select * FROM msg_table WHERE convo = :convo ORDER BY time_stamp ASC")
    fun getConvo(convo: Int): LiveData<List<Msgs>>
```

```
    // Gets all contacts and returns them as a livedata object.
    // Live data objects are update when new data
    // is observed and must be accessed using an observer
    @Query("Select * FROM contacts_table")
    fun getContacts(): LiveData<List<Contacts>>

    // Gets all contacts and returns them as a regular list
    // This is nessacry for when contacts must be
    // checked for validity an observer object would be uneccesary.
    @Query("Select * FROM contacts_table")
    fun instantGetContacts(): List<Contacts>

    // Gets Information about primary user
    @Query("Select * FROM primary_user_table")
    fun getPrimaryUserInfo(): LiveData<List<PrimaryUser>>

    // Lists all Conversations is used in UI
    // for listing all availble messages
    @Query("Select * FROM conversation_table")
    fun getConvos(): LiveData<List<Conversations>>

}
```

Accessing these functions can be done from the database repository which holds functions that abstract multiple data sources. It acts as an intermediate between the DAO and the database View Model. View models are used for when a database need to be interacted with by UI components. Functions here are define just as normal functions. The syntax for declaring repository functions are as follows.

Listing 3: Example of repository definition.

```
class AppRepository(private val dao: DAO) {

    //Inits variables that can me found on start
    val getConvos: LiveData<List<Conversations>> = dao.getConvos()
    val getContacts: LiveData<List<Contacts>> = dao.getContacts()

    // returns all the messages that have the same
    // message ID orderd by time stamp
    fun getConvo(id: Int): LiveData<List<Msgs>>{
        return dao.getConvo(id)
    }

    // Gets all contacts as a list
    fun instantGetContacts(): List<Contacts>{
        return dao.instantGetContacts()
    }

    // Adds a contact to the contact table
    // Uses the suspend keyword to be used in a coroutines
    suspend fun addContact(user: Contacts){
        dao.addContact(user)
    }

    // Adds a primary user to the primary user table
    suspend fun addPrimaryUser(user: PrimaryUser){
        dao.addPrimaryUser(user)
    }

    // Adds a contact to a conversation table
    suspend fun addToConversation(convo: Conversations){
        dao.addToConversation(convo)
    }

    // Adds a message to message table
    suspend fun addMsg(msg: Msgs){
        dao.addMsg(msg)
    }


}
```

The view model uses calls functions from the repository but calls them inside of coroutines that allow for asynchronous access to the database. This is nessacary because the viewmodel is used by UI componets to interacte with the data base so IO calls should be none blocking to prevent slow UI. The app view model is defined by the following code.

Listing 4: Example of Viewmodel definition.

```
// Takes application as context
class AppViewModel(application: Application): AndroidViewModel(application) {
    private val repository: AppRepository

    val getConvos: LiveData<List<Conversations>>
    val getContacts: LiveData<List<Contacts>>

    init {
        val appDao = AppDatabase.getDatabase(application).dao()
        repository = AppRepository(appDao)
        getConvos = repository.getConvos
        getContacts = repository.getContacts
    }

    fun addContact(user: Contacts){
        viewModelScope.launch(Dispatchers.IO){
            repository.addContact(user)

        }
    }

    fun addConvo(convo: Conversations){
        viewModelScope.launch(Dispatchers.IO){
            repository.addToConversation(convo)
        }
    }

    fun getMsgs(id: Int): LiveData<List<Msgs>>{
        return repository.getConvo(id)
    }

    fun instanctGetContacts(): List<Contacts>{
        return repository.instantGetContacts()
    }

}
```

In conclusion these are the necessary database functions to implement the base functions of the messaging app. If additional feature where to be added it is likely there would need to be additional changes to the implementation. For example the current design is centered around single user conversations, If group conversation where to be implemented. There would need to be queries for gather all users in the shared conversation to a list. In the next section the implementation of the HTTP server and how it interacts with the database will be outlined and expanded upon.

The http server was implemented using a kotlin library called "Ktor" [**?**], which is a light kotlin library for creating http servers and clients. Ktor uses coroutines to allow for asynchronous receiving and sending of http requests. For encryption it can use SSL(Secure Sockets Layer) to allow for messages to be sent securely adding an extra layer to the AES256 Encryption that is done when messages are sent. SSL uses private and public certificates to provide asynchronous encryption between servers and clients. Ktor allows for all standard http messages, but the only used for this implementation will be the Post request. The two methods implemented to handle these requests will be the "/txt" http route and the send() function. Below is the Psuedocode for each function.

Listing 5: Psuedocode for /txt.

```
data class Jsonmsg(
        convoEncrypt: String,
        msgEncrypt: String
)

routing{
        // sets the name of the route
        // accessed at http://IP_ADDRESS:PORT/txt
        post("/txt"){

                // When /txt recieves a post request call.recieve
                // retrieves the recieved post data
                lastmsg = call.recieve<Jsonmsg>()

                // Launch a coroutine to prevent the
                // blocking of other messages by IO activities.
                CoroutineScope(Dispatchers.IO).launch {

                        // Retrieves a List of all contacts store by the user
```

```
                        // and then iterates over all contacts to see if a
                        // contact key is able to decrypt the convoEncrypt.
                        // Each decryption is checked to
                        // see if it is the name of a conversation
                        // That currently exists. If it does
                        // it adds a message to the database marked with that messageID

                        val contacts = repository.getInstantContacts()
                        for contact in contacts {
                                convo = decryptAES256(contact.key, lastmsg.convoEncrypt)
                                if(repository.convoexists(convo){
                                        msg = decryptAES256(contact.key, lastmsg.msgEncrpyt)
                                        repository.addmsg(msg(0, convo, msg, timestamp))
                                }
                        }
        }

        }
}
```

Listing 6: Psuedocode for send().

```
        send(msg: String, convoName: String){
                // Find the contact from the contact name
                // and save its keyhash
                contact = repository.findContact(convoName)
                keyhash = contact.key

                // Using AES256 encoding encrpyt the convoName
                // and the msg with the keyhash
                convoEncrypt = AES256(keyhash, convoName)
                msgEncrypt = AES256(keyhash, msg)

                // attempt to post the JSON message using ktor
                // and save the response
                response = http.POST(contact.address){
                        contentType(JSON)
                        setbody(Jsonmsg(convoEncrpyt, msgEncrpyt)
                }

                // Check if the response was OK if not then post
                // "MSG NOT SENT" to approriate chat.
                if(response = http.OK){
                        // Add the primary users msg to the conversation
                        // and add to database
                        repository.addmsg(msg(0, primary.name, convoName, msg))
                } else {
                        repository,addmsg(msg(0, primary.name, convoName,
                        "UNABLE TO SEND TRY AGAIN"))
                }

        }
```

TOR for this project will implement hidden services using Tor Onion Proxy Library a java library [?] developed under Microsoft open source as part of the Thali project [?]. The Thaili project is a group working towards advancing technologies involved in blue tooth and wifi direct networking. The goal of the Tor Onion Proxy Library is to ease the management of TOR proxies and allow each app to act as its own tor proxy. This allows for TOR to be backed into an app instead of relying on third parties like Orbot to create a hidden service. The code for createing the TOR hidden service is relatively simple as little logic is needed to create the service. The code below is sampled from the github for the Tor Onion Proxy Library [?].

Listing 7: Psuedocode for send().

```
String fileStorageLocation = "torfiles";
OnionProxyManager onionProxyManager =
        new AndroidOnionProxyManager(this.getApplicationContext(), fileStorageLocation);
int totalSecondsPerTorStartup = 4 * 60;
int totalTriesPerTorStartup = 5;

// Start the Tor Onion Proxy
if (onionProxyManager.startWithRepeat(totalSecondsPerTorStartup, totalTriesPerTorStartup, true) == false) {
    Log.e("TorTest", "Couldn't start Tor!");
    return;
}
```

```
// Start a hidden service listener
int hiddenServicePort = 80;
int localPort = 9343;
String onionAddress = onionProxyManager.publishHiddenService(hiddenServicePort, localPort);

Socket clientSocket =
        Utilities.socks4aSocketConnection(onionAddress, hiddenServicePort, "127.0.0.1", localPort);
```