Code Review

Project: Ventilate

Group Members: Ryan Porterfield, Jacob Pebworth,
Austin Hoppe, Christopher Hines

Group Name: 4444-Chat_Group-5

Date: 2015-11-12

Contents

Account 1
Chat Room 3
Message 6
Connection Handler 7
Database 8
Database Interface 10
Account Database 11
Chat Room Database 12
Message Database 13
Server 14
Command Parser 14
Account Parser 15
Room Parser 16

Account

```
/*!
 * \brief Check that a user's username and password are valid.
 * \param username A user's account username.
 * \param passwordHash The cryptographic salted hash of the user's password.
 * \return true if the username and password are a valid combination,
otherwise
 * false.
 */
bool Account::authenticateUser(QString& username, QByteArray passwordHash)
{
    AccountDatabase db;
    Account acc = db.find(username);
    return passwordHash == acc.passwordHash;
}

/*!
 * \brief Get the unique ID for the user account.
 * \return  unique ID for the user account.
 */
const QUuid& Account::getUUID() const
{
    return uuid;
}

/*!
 * \brief Get the date and time the user account was created.
 * \return the date and time the user account was created.
 */
const QDateTime& Account::getCreationDate() const
{
    return creationDate;
}

/*!
 * \brief Get the email address used to verify the user account.
 * \return the email address used to verify the user account.
 */
const QString& Account::getEmailAddress() const
{
    return emailAddress;
}

const QByteArray& Account::getPasswordHash() const
{
    return passwordHash;
}

/*!
 * \brief Get the username for the user account.
 * \return the username for the user account.
 */
const QString& Account::getUsername() const
{
    return username;
}

/*!
 * \brief Salt and hash a password so we can store it.
 *
 * Storing plain text or passwords encrypted with a common encryption key is
 * a poor security practice. Anyone who recovers a plain text password file
has
 * access to all user passwords, and if someone recovers both the encrypted
 * table and the encryption key they also have access to all user passwords.
 * By storing salted hashes even if an attacker gets the table of passwords
 * there's no way to find out any individual user's password.
 *
 * \param password String password that is being salted and hashed.
 * \return A cryptographic hash of the user's password.
 */
QByteArray hashPassword(QString& password, QString& username)
{
    QByteArray saltedArray;
    QDataStream out(&saltedArray, QIODevice::WriteOnly);
    out << username;
    out << password;
    return QCryptographicHash::hash(saltedArray,
QCryptographicHash::Sha3_512);
}

/*!
 * \brief Copy operator.
 * \param copy
 * \return this.
 */
Account& Account::operator=(const Account& copy)
{
    uuid = copy.uuid;
    creationDate = copy.creationDate;
    emailAddress = copy.emailAddress;
    passwordHash = copy.passwordHash;
    username = copy.username;
    return *this;
}

/*!
 * \brief Move operator.
 * \param move
 * \return this.
 */
Account& Account::operator=(Account&& move)
{
    uuid = std::move(move.uuid);
    creationDate = std::move(move.creationDate);
    emailAddress = std::move(move.emailAddress);
    passwordHash = std::move(move.passwordHash);
    username = std::move(move.username);
    return *this;
}

/*!
```

```
 * \brief Serialize the Account to a QDataStream.
 * \param out QDataStream the Account is being serialized to.
 * \param account the Account being serialized.
 * \return a modified version of out with the account in it.
 */
QDataStream& operator<<(QDataStream& out, const Account& account)
{
    out << account.uuid;
    out << account.username;
    out << account.creationDate;
    out << account.passwordHash;
    out << account.emailAddress;
    return out;
}

/*!
 * \brief Get an account that was serialized.
 * \param in
 * \param account
 * \return
 */
QDataStream& operator>>(QDataStream& in, Account& account)
{
    in >> account.uuid;
    in >> account.username;
    in >> account.creationDate;
    in >> account.passwordHash;
    in >> account.emailAddress;
    return in;
}
```

Chat Room

```
void ChatRoom::addMessage(const Message& message)
{
    messages.append(message);
}

void ChatRoom::addMessages(const QList<Message>& messages)
{
    for (Message msg : messages)
        addMessage(msg);
}

void ChatRoom::addModerator(const QString& mod)
{
    ModDatabase db;
    moderators.append(mod);
    db.add(mod, uuid);
}

void ChatRoom::addModerators(const QList<QString>& mods)
{
    for (QString mod : mods)
```

```
        addModerator(mod);
}

void ChatRoom::addUser(const QString& user)
{
    UserDatabase db;
    users.append(user);
    db.add(user, uuid);
}

void ChatRoom::addUsers(const QList<QString>& users)
{
    for (QString user : users)
        addUser(user);
}

void ChatRoom::getHistory()
{
    MessageDatabase db;
    QList<Message> history = db.getMessages(uuid, messages.size());
    QList<Message>::iterator iter = history.end() - 1;
    for (; iter != history.begin(); --iter)
        messages.prepend(*iter);
}

QString ChatRoom::getMessages()
{
    QString all_msgs = "";
    for (Message msg : messages) {
        all_msgs.append(serializeMessage(msg));
        all_msgs.append("\n");
    }
    return all_msgs;
}

const QList<QString>& ChatRoom::getModerators() const
{
    return moderators;
}

const QString& ChatRoom::getName() const
{
    return name;
}

const QString& ChatRoom::getOwner() const
{
    return owner;
}

const QUuid& ChatRoom::getUUID() const
{
    return uuid;
}

const QList<QString>& ChatRoom::getUsers() const
{
    return users;
}

void ChatRoom::removeModerator(const QString& mod)
{
    ModDatabase db;
    moderators.removeOne(mod);
    db.remove(mod, uuid);
}

void ChatRoom::removeUser(const QString& user)
{
    UserDatabase db;
    users.removeOne(user);
    db.remove(user, uuid);
}

QString ChatRoom::serializeMessage(const Message& message)
{
    QString msg_str = "[" + message.getTimeStamp().toString() + "] ";
    msg_str.append(message.getUsername());
    msg_str.append(": ");
    msg_str.append(message.getMessage());
    return msg_str;
}


ChatRoom& ChatRoom::operator=(const ChatRoom& copy)
{
    QObject::setParent(copy.parent());
    uuid = copy.uuid;
    owner = copy.owner;
    name = copy.name;
    moderators = copy.moderators;
    users = copy.users;
    messages = copy.messages;
    return *this;
}

ChatRoom& ChatRoom::operator=(ChatRoom&& move)
{
    QObject::setParent(move.parent());
    move.setParent(nullptr);
    uuid = std::move(move.uuid);
    owner = std::move(move.owner);
    name = std::move(move.name);
    moderators = std::move(move.moderators);
    users = std::move(move.users);
    messages = std::move(move.messages);
    return *this;
}

QDataStream& operator<<(QDataStream& out, const ChatRoom& room)
{
    out << room.getUUID();
    out << room.getName();
    out << room.getOwner();
    return out;
}

QDataStream& operator>>(QDataStream& in, ChatRoom& room)
{
    in >> room.uuid;
    in >> room.name;
    in >> room.owner;
    return in;
}
```

Message

```
QString Message::getFormattedMessage() const
{
    QString msgstr = getHeader();
    msgstr = msgstr.append(message);
    return msgstr;
}

QString Message::getHeader() const
{
    QString msgstr("[");
    msgstr.append(timestamp.time().toString()).append("] ").append(username);
    return msgstr.append(": ");
}


QString Message::getSanitizedMessage() const
{
    QString msgstr = getHeader();
    QString clone(message);
    clone.replace(QChar('\\'), QString("\\\\"));
    return msgstr.append(clone);
}


const QString& Message::getMessage() const
{
    return message;
}

const QUuid& Message::getRoomID() const
{
    return roomID;
}

const QDateTime& Message::getTimeStamp() const
{
    return timestamp;
}

const QString& Message::getUsername() const
{
    return username;
}

const QUuid& Message::getUUID() const
{
    return uuid;
}
```

Connection Handler

```
/*!
 * \brief Called when a client disconnects from the Server.
 */
void ConnectionHandler::disconnected()
{
    qDebug() << socketDescriptor << " Disconnected";
    Server *server = static_cast<Server*>(this->parent());
    server->disconnectClient(this);
    socket->deleteLater();
    exit(0);
}

const QHostAddress& ConnectionHandler::getHostAddress() const
{
    return std::move(QHostAddress(socket->peerAddress()));
}

/*!
 * \brief Connect to a client.
 */
void ConnectionHandler::run()
{
    qDebug() << "Opened a new ConnectionHandler";
    socket = new QTcpSocket();
    if (!socket->setSocketDescriptor(socketDescriptor)) {
        emit error(socket->error());
        return;
    }
    qDebug() << "Client address: " << socket->peerAddress();
    connect(socket, SIGNAL(readyRead()), this, SLOT(readyRead()),
Qt::DirectConnection);
    connect(socket, SIGNAL(disconnected()), this, SLOT(disconnected()));
    qDebug() << "Connected to " << socketDescriptor;

    exec();
}

void ConnectionHandler::readyRead()
{
    static qint16 blockSize = 0;
    QDataStream in(socket);
    in.setVersion(QDataStream::Qt_5_0);
    if (blockSize == 0) {
        if (socket->bytesAvailable() < (int) sizeof(quint16))
```

```
            return;
        in >> blockSize;
    }
    if (socket->bytesAvailable() < blockSize)
        return;
    blockSize = 0;
    Server *server = static_cast<Server*>(parent());
    server->onClientRequest(*this, in);
}

/**
 * @brief Sends a message to the client.
 * @param data A preformatted message ready to be written directly to the
client.
 */
void ConnectionHandler::sendToClient(QByteArray data) const
{
    qDebug() << "Sending data: " << data;
    socket->write(data);
}

void ConnectionHandler::write(QString data) const
{
    QByteArray block;
    QDataStream out(&block, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_5_0);
    // Reserve space for size of block
    out << (quint16) 0;
    out << data;
    // Seek back to begining of block
    out.device()->seek(0);
    // Insert size of block at beginning
    out << (quint16) (block.size() - sizeof(quint16));
    sendToClient(block);
}
```

Database

```
void Database::createAccountDB(QSqlDatabase& db)
{
    qDebug() << "Creating account database";
    db.transaction();
    QSqlQuery query(db);
    query.prepare("CREATE TABLE IF NOT EXISTS " + ACCOUNT_TABLE + "("
               + ID_KEY + " BLOB NOT NULL UNIQUE PRIMARY KEY, "
               + DATE_KEY + " DATETIME NOT NULL, "
               + EMAIL_KEY + " TEXT NOT NULL UNIQUE, "
               + PASSWORD_KEY + " BLOB NOT NULL, "
               + NAME_KEY + " TEXT NOT NULL UNIQUE);");
    runQuery(query);
    db.commit();
}

void Database::createMessageDB(QSqlDatabase& db)
```

```
{
    qDebug() << "Creating message database";
    db.transaction();
    QSqlQuery query(db);
    query.prepare("CREATE TABLE IF NOT EXISTS " + MESSAGE_TABLE + "("
                  + ID_KEY + " BLOB NOT NULL UNIQUE PRIMARY KEY, "
                  + ROOM_KEY + " BLOB NOT NULL, "
                  + DATE_KEY + " DATETIME NOT NULL, "
                  + MESSAGE_KEY + " TEXT NOT NULL, "
                  + NAME_KEY + " TEXT NOT NULL, "
                  + "FOREIGN KEY(" + ROOM_KEY + ") REFERENCES "
                  + ROOM_TABLE + "(" + ID_KEY + ") ON UPDATE CASCADE, "
                  + "FOREIGN KEY(" + NAME_KEY + ") REFERENCES "
                  + ACCOUNT_TABLE + "(" + NAME_KEY + ") ON UPDATE
CASCADE);");
    runQuery(query);
    db.commit();
}

void Database::createModeratorDB(QSqlDatabase& db)
{
    qDebug() << "Creating moderator database";
    db.transaction();
    QSqlQuery query(db);
    query.prepare("CREATE TABLE IF NOT EXISTS " + MOD_TABLE + "("
        + NAME_KEY + " TEXT NOT NULL, "
        + ID_KEY + " BLOB NOT NULL, "
        + "PRIMARY KEY(" + NAME_KEY + ", " + ID_KEY + "), "
        + "FOREIGN KEY(" + NAME_KEY + ") REFERENCES "
        + ACCOUNT_TABLE + "(" + NAME_KEY + ") ON UPDATE CASCADE, "
        + "FOREIGN KEY(" + ID_KEY + ") REFERENCES "
        + ROOM_TABLE + "(" + ID_KEY + ") ON UPDATE CASCADE);"
    );
    runQuery(query);
    db.commit();
}

void Database::createRoomDB(QSqlDatabase& db)
{
    qDebug() << "Creating room database";
    db.transaction();
    QSqlQuery query(db);
    query.prepare("CREATE TABLE IF NOT EXISTS " + ROOM_TABLE + "("
        + ID_KEY + " BLOB NOT NULL UNIQUE PRIMARY KEY, "
        + OWNER_KEY + " TEXT NOT NULL, "
        + NAME_KEY + " TEXT NOT NULL, "
        + "FOREIGN KEY(" + OWNER_KEY + ") REFERENCES "
        + ACCOUNT_TABLE + "(" + NAME_KEY + ") ON UPDATE CASCADE);"
    );
    runQuery(query);
    db.commit();
}

void Database::createUserDB(QSqlDatabase& db)
{
    qDebug() << "Creating user database";
    db.transaction();
    QSqlQuery query(db);
    query.prepare("CREATE TABLE IF NOT EXISTS " + USER_TABLE + "("
        + NAME_KEY + " TEXT NOT NULL, "
        + ID_KEY + " BLOB NOT NULL, "
        + "PRIMARY KEY(" + NAME_KEY + ", " + ID_KEY + "), "
        + "FOREIGN KEY(" + NAME_KEY + ") REFERENCES "
        + ACCOUNT_TABLE + "(" + NAME_KEY + ") ON UPDATE CASCADE, "
        + "FOREIGN KEY(" + ID_KEY + ") REFERENCES "
        + ROOM_TABLE + "(" + ID_KEY + ") ON UPDATE CASCADE);"
    );
    runQuery(query);
    db.commit();
}

void Database::init()
{
    QSqlDatabase db(QSqlDatabase::addDatabase("QSQLITE", DATABASE_NAME));
    openDB(db);
    createAccountDB(db);
    createRoomDB(db);
    createMessageDB(db);
    createModeratorDB(db);
    createUserDB(db);
    db.close();
    QSqlDatabase::removeDatabase(DATABASE_NAME);
}

void Database::openDB(QSqlDatabase& db)
{
    QString path = QCoreApplication::applicationDirPath() +
"/ventilate.sqlite";
    qDebug() << "Database path: " << path;
    db.setDatabaseName(path);
    if (!db.open()) {
        qDebug() << "DATABASE NOT OPENED: " << db.lastError().text();
        qDebug() << "";
    } else {
        qDebug() << "DATABASE OPENED";
        qDebug() << "";
    }
}
```

Database Interface

```
/*!
 * \brief Build an object from a database query.
 *
 * This templated pure virtual function requires concrete sub-classes to
 * build an object, most likely an Account, ChatRoom, or Message that was
 * stored in the database.
 *
 * \param query Results of the SQL query.
 * \return An object from the database.
 */
```

```
virtual T buildFromQuery(const QSqlQuery& query) const = 0;
```

Account Database

```
bool AccountDatabase::add(const Account& elem)
{
    qDebug() << "Adding row to table: " << elem.getUsername();
    db.transaction();
    QSqlQuery query(db);
    query.prepare("INSERT INTO " + ACCOUNT_TABLE +
                  "(" + ID_KEY + ", " + DATE_KEY  + ", "
                  + EMAIL_KEY + ", "  + PASSWORD_KEY + ", " + NAME_KEY + ")"
                  + " VALUES(?, ?, ?, ?, ?);");
    query.addBindValue(elem.getUUID());
    query.addBindValue(elem.getCreationDate());
    query.addBindValue(elem.getEmailAddress());
    query.addBindValue(elem.getPasswordHash());
    query.addBindValue(elem.getUsername());
    bool flag = runQuery(query);
    db.commit();
    return flag;
}


Account AccountDatabase::buildFromQuery(const QSqlQuery& query) const
{
    QUuid id = query.value(ID_KEY).toByteArray();
    QDateTime date = query.value(DATE_KEY).toDateTime();
    QString email = query.value(EMAIL_KEY).toString();
    QByteArray password = query.value(PASSWORD_KEY).toByteArray();
    QString username = query.value(NAME_KEY).toString();
    return std::move(Account(id, username, date, password, email));
}


Account AccountDatabase::find(const QUuid& id)
{
    return std::move(DatabaseInterface::find(id, ACCOUNT_TABLE));
}

Account AccountDatabase::find(const QString &username)
{
    qDebug() << "Finding row in table: " << username;
    db.transaction();
    QSqlQuery query(db);
    query.prepare("SELECT * FROM " + ACCOUNT_TABLE
                  + " WHERE " + NAME_KEY + " = ?;");
    query.addBindValue(username);
    runQuery(query);
    query.first();
    db.commit();
    return std::move(buildFromQuery(query));
}
```

```
QList<Account> AccountDatabase::getAll()
{
    return std::move(DatabaseInterface::getAll(ACCOUNT_TABLE));
}

bool AccountDatabase::remove(const Account& elem)
{
    return DatabaseInterface::remove(elem.getUUID(), ACCOUNT_TABLE);
}
```

Chat Room Database

```
bool ChatRoomDatabase::add(const ChatRoom &elem)
{
    qDebug() << "Adding row to table" << elem.getName();
    db.transaction();
    QSqlQuery query(db);
    query.prepare("INSERT INTO " + ROOM_TABLE + "(" + ID_KEY + ", "
                  + OWNER_KEY + ", "  + NAME_KEY + ")" + "
VALUES(?, ?, ?);");
    query.addBindValue(elem.getUUID());
    query.addBindValue(elem.getOwner());
    query.addBindValue(elem.getName());
    bool flag = runQuery(query);
    db.commit();
    return flag;
}

ChatRoom ChatRoomDatabase::buildFromQuery(const QSqlQuery &query) const
{
    QUuid id = query.value(ID_KEY).toByteArray();
    QString owner = query.value(OWNER_KEY).toString();
    QString name = query.value(NAME_KEY).toString();
    return std::move(ChatRoom(id, owner, name));
}

ChatRoom ChatRoomDatabase::find(const QUuid &roomID)
{
    ChatRoom room = DatabaseInterface::find(roomID, ROOM_TABLE);
    MessageDatabase mdb;
    ModDatabase modb;
    UserDatabase udb;
    QList<Message> messages = mdb.getMessages(room.getUUID(), 0);
    QList<QString> users = udb.get(roomID);
    QList<QString> mods = modb.get(roomID);
    room.addMessages(messages);
    room.addModerators(mods);
    room.addUsers(users);
    return std::move(room);
}

QList<ChatRoom> ChatRoomDatabase::getAll()
{
    return std::move(DatabaseInterface::getAll(ROOM_TABLE));
}

bool ChatRoomDatabase::remove(const ChatRoom &elem)
{
    return DatabaseInterface::remove(elem.getUUID(), ROOM_TABLE);
}
```

Message Database

```
bool MessageDatabase::add(const Message& elem)
{
    qDebug() << "Adding row to table" << elem.getMessage();
    db.transaction();
    QSqlQuery query(db);
    query.prepare("INSERT INTO " + MESSAGE_TABLE +
                  "(" + ID_KEY + ", " + ROOM_KEY + ", " + DATE_KEY  + ", "
                  + MESSAGE_KEY + ", "  + NAME_KEY + ")"
                  + " VALUES(?, ?, ?, ?, ?);");
    query.addBindValue(elem.getUUID());
    query.addBindValue(elem.getRoomID());
    query.addBindValue(elem.getTimeStamp());
    query.addBindValue(elem.getMessage());
    query.addBindValue(elem.getUsername());
    bool flag = runQuery(query);
    db.commit();
    return flag;
}

Message MessageDatabase::buildFromQuery(const QSqlQuery &query) const
{
    QUuid id = query.value(ID_KEY).toByteArray();
    QUuid room = query.value(ROOM_KEY).toByteArray();
    QDateTime date = query.value(DATE_KEY).toDateTime();
    QString message = query.value(MESSAGE_KEY).toString();
    QString username = query.value(NAME_KEY).toString();
    return std::move(Message(id, room, date, username, message));
}

Message MessageDatabase::find(const QUuid& id)
{
    return std::move(DatabaseInterface::find(id, MESSAGE_TABLE));
}

QList<Message> MessageDatabase::getAll()
{
    return std::move(DatabaseInterface::getAll(MESSAGE_TABLE));
}

QList<Message> MessageDatabase::getMessages(const QUuid& roomID, quint32
start)
{
    qDebug() << "Getting Messages from database";
    db.transaction();
    QSqlQuery query(db);
    query.prepare("SELECT * FROM " + MESSAGE_TABLE + " WHERE " + ROOM_KEY
                  + " = ? ORDER BY " + DATE_KEY + " DESC LIMIT " +
RETURN_RANGE + " OFFSET "
                  + QString::number(start) + ";");
    query.addBindValue(roomID);
    runQuery(query);
    db.commit();
    QList<Message> list;
    while (query.next())
        list.append(buildFromQuery(query));
    return std::move(list);
}

bool MessageDatabase::remove(const Message& elem)
{
    return DatabaseInterface::remove(elem.getUUID(), MESSAGE_TABLE);
}
```

Server

```
/*!
 * \brief Handle requests from the clients.
 *
 * This function gets called any time a ConnectionHandler recieves a request
 * from a client over the network. Some preliminary command parsing is done,
 * then the handler and command stream are passed off to an appropriate
 * CommandParser sub-class to handle the command.
 *
 * \param handler Reference to the ConnectionHandler that recieved the
 * request.
 * \param request QDataStream that the handler read in from the network.
 */
void Server::onClientRequest(const ConnectionHandler& handler, QDataStream&
stream)
{
    QString cmd;
    stream >> cmd;
    qDebug() << "Got string: " << cmd << " from stream";
    if (cmd == CommandParser::ROOM)
        roomParser.parse(handler, stream);
    else if (cmd == CommandParser::ACCOUNT || cmd == CommandParser::LOGIN)
        accountParser.parse(handler, stream);
    else if (cmd == CommandParser::PEER)
        peerParser.parse(handler, stream);
    else if (cmd == CommandParser::PASSWORD)
        passwordParser.parse(handler, stream);
    // Drop incorrectly formatted requests
}
```

Command Parser

```
/*!
```

```
 * \brief Parse an incoming command from a client.
 *
 * This pure virtual function must be implemented by all concrete
 * sub-classes. This function handles extended parsing of sub-commands.
 *
 * \param handler Reference to the ConnectionHandler that recieved the
 * request.
 * \param request QDataStream that the handler read in from the network.
 */
virtual void parse(const ConnectionHandler& handler, QDataStream& stream) = 0;
```

Account Parser

```
void AccountParser::create(const ConnectionHandler& handler, QDataStream&
stream)
{
    QUuid uuid;
    QString username, email;
    QDateTime time;
    QByteArray phash;
    stream >> uuid >> username >> time >> phash >> email;
    Account acc(uuid, username, time, phash, email);
    AccountDatabase db;
    if (db.add(acc))
        handler.write(ACCEPT);
    else
        handler.write(REJECT + " " + GENERIC_ERROR);
}

void AccountParser::login(const ConnectionHandler& handler, QDataStream&
stream)
{
    QString username;
    QByteArray phash;
    stream >> username >> phash;
    if (Account::authenticateUser(username, phash))
        handler.write(ACCEPT);
    else
        handler.write(REJECT + " " + INVALID_PASSWORD);
}

void AccountParser::parse(const ConnectionHandler& handler, QDataStream&
stream)
{
    QString cmd;
    stream >> cmd;
    if (cmd == LOGIN)
        login(handler, stream);
    else if (cmd == CREATE)
        create(handler, stream);
    else if (cmd == DELETE)
        remove(handler, stream);
}
```

```
void AccountParser::remove(const ConnectionHandler& handler, QDataStream&
stream)
{
    QString username;
    QByteArray phash;
    stream >> username >> phash;
    if (!Account::authenticateUser(username, phash)) {
        handler.write(REJECT + " " + INVALID_PASSWORD);
        return;
    }
    AccountDatabase db;
    Account acc = db.find(username);
    if (db.remove(acc))
        handler.write(ACCEPT);
    else
        handler.write(REJECT + " " + GENERIC_ERROR);
}
```

Room Parser

```
void RoomParser::add(const ConnectionHandler& handler, QDataStream& stream)
{
    QString username;
    QUuid roomID;
    stream >> username >> roomID;
    ChatRoomDatabase db;
    ChatRoom room = db.find(roomID);
    room.addUser(username);
    handler.write(ACCEPT);
}

void RoomParser::create(const ConnectionHandler& handler, QDataStream&
stream)
{
    QString roomName;
    QString owner;
    stream >> roomName >> owner;
    ChatRoom room(owner, roomName);
    ChatRoomDatabase db;
    if (db.add(room))
        handler.write(ACCEPT);
    else
        handler.write(REJECT);
}

void RoomParser::history(const ConnectionHandler& handler, QDataStream&
stream)
{
    QString cmd;
    stream >> cmd;
    if (cmd != LIST)
        return;
    QUuid roomID;
    quint32 offset;
    stream >> roomID >> offset;
    MessageDatabase db;
    QList<Message> history = db.getMessages(roomID, offset);
    QString hisstr = ROOM + " " + HISTORY + " ";
    for (Message msg : history)
        hisstr = hisstr.append(msg.getMessage()).append(LIST_SEPARATOR);
    handler.write(hisstr);
}

void RoomParser::join(const ConnectionHandler& handler, QDataStream& stream)
{
    QUuid roomID;
    QString username;
    stream >> roomID >> username;
    ChatRoomDatabase db;
    ChatRoom room = db.find(roomID);
    room.addUser(username);
    handler.write(ACCEPT);
}

void RoomParser::leave(const ConnectionHandler& handler, QDataStream& stream)
{
    QUuid roomID;
    QString username;
    stream >> roomID >> username;
    ChatRoomDatabase db;
    ChatRoom room = db.find(roomID);
    room.removeUser(username);
    handler.write(ACCEPT);
}

void RoomParser::list(const ConnectionHandler& handler, QDataStream& stream)
{
    QString cmd;
    stream >> cmd;
    if (cmd != LIST)
        return;
    ChatRoomDatabase db;
    QList<ChatRoom> rooms = db.getAll();
    QString roomstr = ROOM + " " + LIST + " ";
    for (ChatRoom room : rooms)
        roomstr = roomstr.append(room.getName()).append(LIST_SEPARATOR);
    handler.write(roomstr);
}

void RoomParser::message(QDataStream& stream)
{
    QUuid messageID;
    QUuid roomID;
    QDateTime time;
    QString sender;
    QString message;
    stream >> messageID >> roomID >> time >> sender >> message;
    Message msg(messageID, roomID, time, sender, message);
    MessageDatabase db;
    db.add(msg);
    propogateMessage(msg);
```

```
}

void RoomParser::parse(const ConnectionHandler& handler, QDataStream& stream)
{
    QString cmd;
    stream >> cmd;
    /* Try to order these in most common first so we're not spending excess
     * time doing unnecessary string comparisons. */
    if (cmd == MESSAGE)
        message(stream);
    else if (cmd == JOIN)
        join(handler, stream);
    else if (cmd == LEAVE)
        leave(handler, stream);
    else if (cmd == ADD)
        add(handler, stream);
    else if (cmd == HISTORY)
        history(handler, stream);
    else if (cmd == LIST)
        list(handler, stream);
    else if (cmd == CREATE)
        create(handler, stream);
    else if (cmd == DELETE)
        remove(handler, stream);
    else if (cmd == MODE)
        mode(handler, stream);
}

void RoomParser::propogateMessage(const Message& message)
{
    for (ConnectionHandler* handler : clientList) {
        QString command(ROOM);
        command.append(" ").append(MESSAGE).append(" ");
        command.append(message.getSanitizedMessage());
        handler->write(command);
    }
}

void RoomParser::remove(const ConnectionHandler& handler, QDataStream&
stream)
{
    QUuid roomID;
    QString username;
    QByteArray phash;
    stream >> roomID >> username >> phash;
    if (!Account::authenticateUser(username, phash)) {
        handler.write(REJECT + " " + INVALID_PASSWORD);
        return;
    }
    ChatRoomDatabase db;
    ChatRoom room = db.find(roomID);
    if (db.remove(room))
        handler.write(ACCEPT);
    else
        handler.write(REJECT + " " + GENERIC_ERROR);
}
```