CS 5617 Software Engineering

Content Module Specification

By Sughandhan S, Team Lead

Table of Contents:

1. Overview 1
2. Content Team 2
3. Goals/Objectives..... 2
4. Roles and Individual Objectives..... 3
5. Design Patterns 4
6. Interfaces and Class Diagrams 5
7. UML Diagram 10
8. Activity Diagram 10
9. UX Design 14
10. Summary and Conclusion 15

Overview

PlexShare is a Lab Session Monitoring application where students and instructors can communicate and collaborate by sharing screens, content, and a digital whiteboard in pursuit of their assignment. This specification document serves as the ground truth for what the content team of PlexShare aims to achieve. The content module is an essential part of the application which involves messaging and file sharing among clients. This module is divided into three main components: UX, Client-side content and Server-side content components.

- The UX component deals with designing the user interface, implementing required user actions and displaying messages/files.
- The Client-side content component handles the business logic and processing of various events sent to / received from the UX component. It communicates with the server-side content component via a networking module for storing data.

- Finally, the Server-side content component is responsible for storing and fetching chats and files from the database, processing the received requests and sending them to the client-side content component via the networking module.

Content Team

1. **Lead: Sughandhan S, 111901049**
2. **Members:**
    a. **Narvik Nandan, 111901035**
    b. **Anurag Jha, 111901010**

Goals/Objectives

1. **Sharing and receiving messages:** Users should be able to do the following
    a. **React to Messages**
    b. **Edit their Messages**
    c. **Delete their Messages**
    d. **Reply to Messages**
    e. **Star Messages**
2. **Sharing and receiving documents:** Users should be able to share and download files of PDF, Docx, image formats etc.
3. **UX Design:**
    a. Design MESSAGE Box with SEND icon for the user to type their messages and upload files to share with all the subscribers(broadcast)

b. Displaying the messages/files sent by a user along with their username in the chat window as a chat bubble.

c. Implement React functionality over the chat bubble to allow users to react with an emoji.

d. Implement Edit functionality over the chat bubble to let only the user who has sent the corresponding message edit it

e. Implement Delete functionality over the chat bubble to let only the user who has sent the corresponding message delete it.

f. Implement Reply functionality over the chat bubble for a user to reply to any specific message in the chat window. In the Reply chat message's bubble, there will be a preview of the message being replied to at the top.

g. Implement the Star functionality over the chat bubble for the users to mark the following message as IMPORTANT. These messages will be used by the dashboard team to include them in their summary.

All our features will be available in the LAB mode and EXAM mode. The key difference is, when EXAM mode is enabled, students' messages will only be visible to the instructor. If the instructor types in chat by default it will be a broadcast message. If the instructor replies to a specific message, it will be only visible to the user of that message.

Roles and Individual Objectives:

1. **UX (Sughandhan S)**
   - 1.1. Design the chat window
   - 1.2. Handling the sending and receiving of chats and files from the content module
   - 1.3. Implement the following features for the session users:
     - 1.3.1. Reply to message

1.3.2. React to message

1.3.3. Edit message

1.3.4. Delete Message

1.3.5. Star Message

2. Content Client (Narvik Nandan)

2.1. Implementation of required interfaces and classes to communicate with other modules. Also includes publisher-subscriber design pattern between client-side content component (publisher) and clients on the UX component (subscribers)

2.2. Receiving messages from UX component, processing them and sending them to network module.

2.3. Handling various user actions such as editing and deleting messages, replying to specific chats, uploading and downloading files, etc.

2.4. Subscribing to network module , processing the received messages and informing the subscribers (clients subscribed to client-side content component) about the received messages.

3. Content Server Management (Anurag Jha)

3.1. Subscribing to the network module to listen to the message objects.

3.2. Deserializing the message objects received and separating them into 2 sets.

3.3. In case of a message, add it to the database.

3.4. In case of a file, store it in a database, and allow clients to download whenever necessary.

3.5. Create a message object and convert into XML string, and then either broadcast it or send it privately based on the type of request.

3.6. Handle sending data to dashboard module upon request from the dashboard.

Design Patterns

Singleton Pattern:

We use this pattern as we only need one instance of our Content Client Manager and one instance of our Content Server Manager. The factories on our client and server depend on this design.

- Advantages
    - Easier to hide dependencies.
    - Easy to maintain.
    - Interfaces can be implemented.
    - Single Point Access.

Factory Design Pattern: We use this design pattern in our Factory Content Server and also in the client side. We utilise this pattern to implement singleton factory design pattern.

Publisher Subscriber: This pattern allows the client side to publish the messages over the network for the content server. The content server subscribes to the network module to listen to these messages and take the appropriate action. The UX module subscribes to the client content to receive necessary message and action. The dashboard subscribes to the content server to get messages for summary

- Advantages
    - Publisher and subscriber do not know each other. Hence reduces complexity,
    - Easy to maintain as changes on one side have minimal impact on the other side.

Interfaces and Classes Diagrams

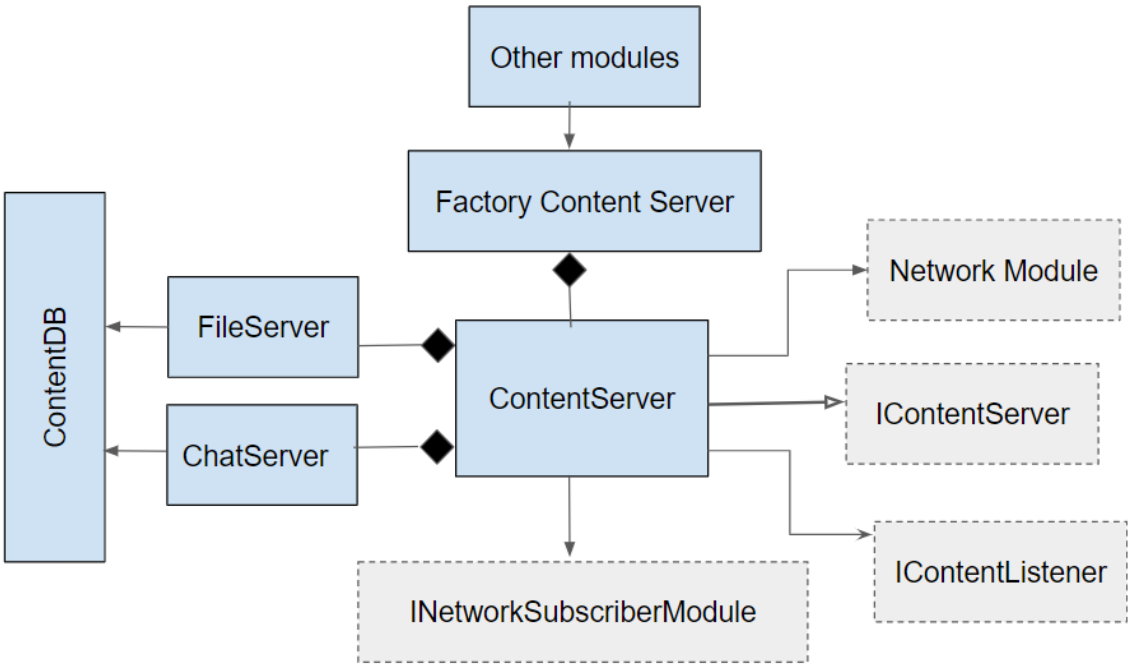
- **Content Client(By Narvik Nandan, 111901035)**:

```
public interface IContentClient {
  /// <summary>
  /// Sends chat or file data to clients
  /// </summary>
  /// <param name="contentData">Data to send, can be chat or file content data.</param>
  void ClientSendData(SendContentData contentData);

  /// <summary>
  /// Edits a chat message
  /// <summary>
  /// <param name="messageID">Chat message ID to be updated</param>
  /// <param name="messageContent">Chat message content to be updated</param>
  void ClientEditChat(int messageID, string messageContent);

  /// <summary>
  /// Deletes the chat message
  /// </summary>
  /// <param name="messageID">Chat message ID to be deleted</param>
  void ClientDeleteChat(int messageID);

  /// <summary>
  /// Stars a message (will be included in the Dashboard summary)
  /// </summary>
  /// <param name="messageID">Chat message ID to be starred</param>
  void ClientStarChat(int messageID);

  /// <summary>
  /// React to a message with emojis
  /// </summary>
  /// Have to decide on the parameters and implementation
```

```
    void ClientReact();

    /// <summary>
    /// Downloads file on the client machine in the specified path
    /// </summary>
    /// <param name="messageId">File message ID to be downloaded</param>
    /// <param name="path">Path in which file will be downloaded</param>
    void ClientDownload(int messageID, string path);

    /// <summary>
    /// Subscribes to content module for listening to received messages
    /// </summary>
    /// <param name="subscriber">An instance of IContentListener interface</param>
    void ClientSubscribe(IContentListener subscriber);

    /// <summary>
    /// Gets the reply message ID
    /// </summary>
    /// <param name="replyID">Reply message ID</param>
    void ClientGetReplyID(int replyID);
}
```

```
public interface IContentListener{
    /// <summary>
    /// Handles messages received by content module
    /// </summary>
    /// <param name="contentData">Received content data</param>
    void OnMessage(ReceiveContentData contentData)

    /// <summary>
    /// Handles all messages sent to client
    /// </summary>
    /// have to decide on parameters and functionality
    void OnAllMessages()
}
```

- **Content Server(By Anurag Jha, 111901010)**:



An interface contains definitions for a group of related functionalities that a non-abstract class or a struct must implement. Static methods may be defined by an interface. Server side contains an IContentServer interface. The functions supported by the interface are:

- **ServerSubscribe(IContentListener listener)** : This function uses the publisher-subscriber design pattern. It is mainly intended for usage by the dashboard module. When the dashboard team requests for the starred messages, they will need to provide an object that implements the IContentListener interface.
- **ServerGetMessages()** : This function can be used to fetch all the messages that have been sent. It basically returns a list. The list can further be filtered based on the time of creation.

- **ServerSendMessages(userID)**: This function sends all the messages that have been sent till now to the new user that has just connected to the server.

- **UX**:

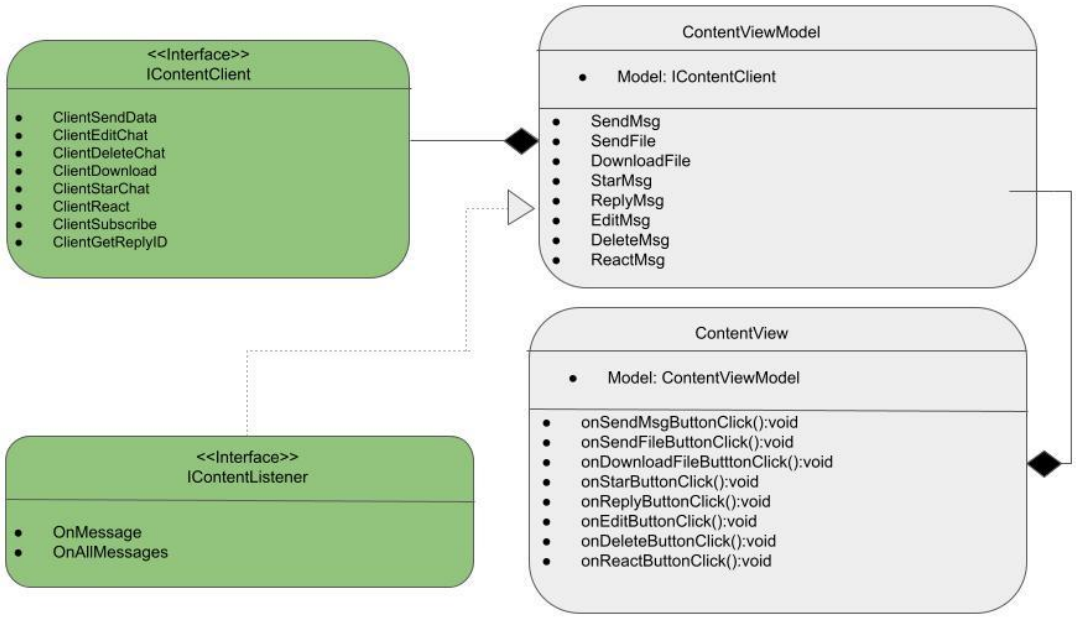
  ContentViewModel:

  - The **ContentViewModel** class implements Content Module's **IContentListener** interface. This object is passed to the **CSubscribe** function of **IContentClient** to receive messages/files from the content client
    - Receiving messages or files in the form of a **ReceiveMessageData** object requires the use of the **OnMessage** function. The fields for the *event type, message type, message, message id, sender, recipient, reply thread, sent time,* and *isStarred* are all included in this object.
    - All messages and files transmitted up until this point are collected by the **OnAllMessages** function. This function makes sure that all previous messages may be viewed by clients who have either joined or rejoined.
  - The **CSend** function provided by **IContentClient** will be used to send a message to the content module. The message will be deserialised and constructed as an object which has fields which specifies its *type(chat or file), message, receiver ids, message id, delete flag*. The *delete flag* is used to denote if the message has been deleted and hence no further changes can be made. The *message id* will be newly generated **CEditChat** function is similar to send but this function's object parameter will have the *message id* field as the id of the message it will overwrite. **CDeleteChat** is the same as **CEditChat** but the subtle difference is the overwritten message is *"Message Deleted."* and the delete flag will be set.
  - The **IContentClient** interface's **CDownload** function will be used for downloading files. It accepts two arguments—the message id and the directory to save the file in. The

content module can therefore download the file from the server and save it to the specified file location.

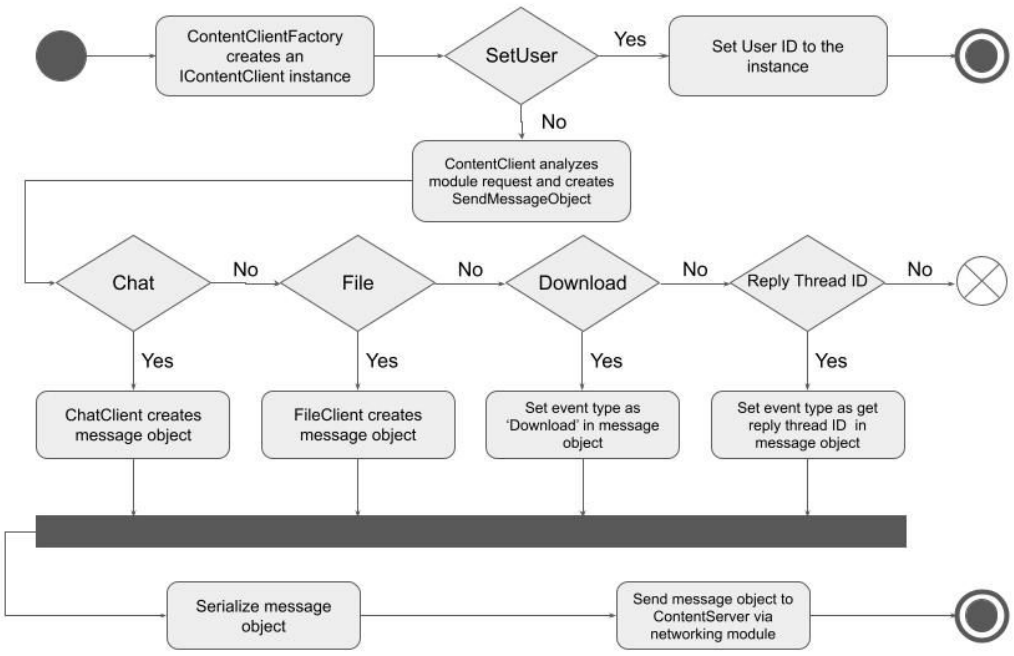
- **CStarChat** function of the **IContentClient** interface takes in the message id of the starred message and sends it to the content module so that it will be included in the summary generated by the dashboard module

UML Diagram

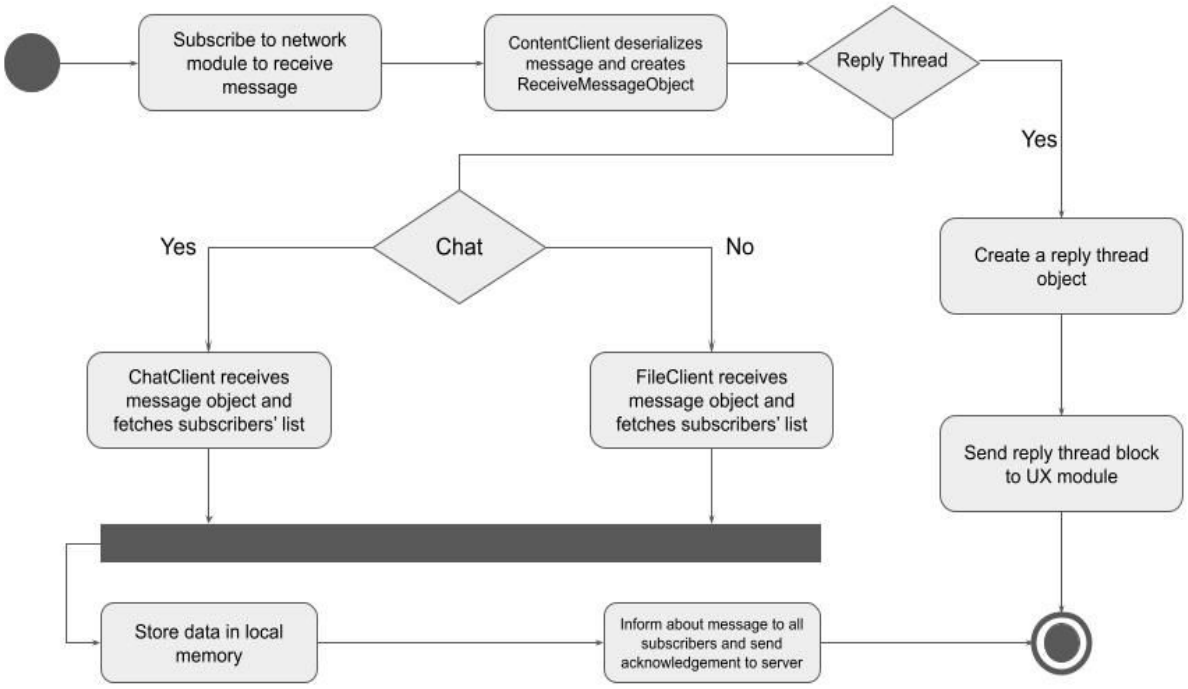


Activity Diagrams

- **Content Client Side(By Narvik Nandan):**

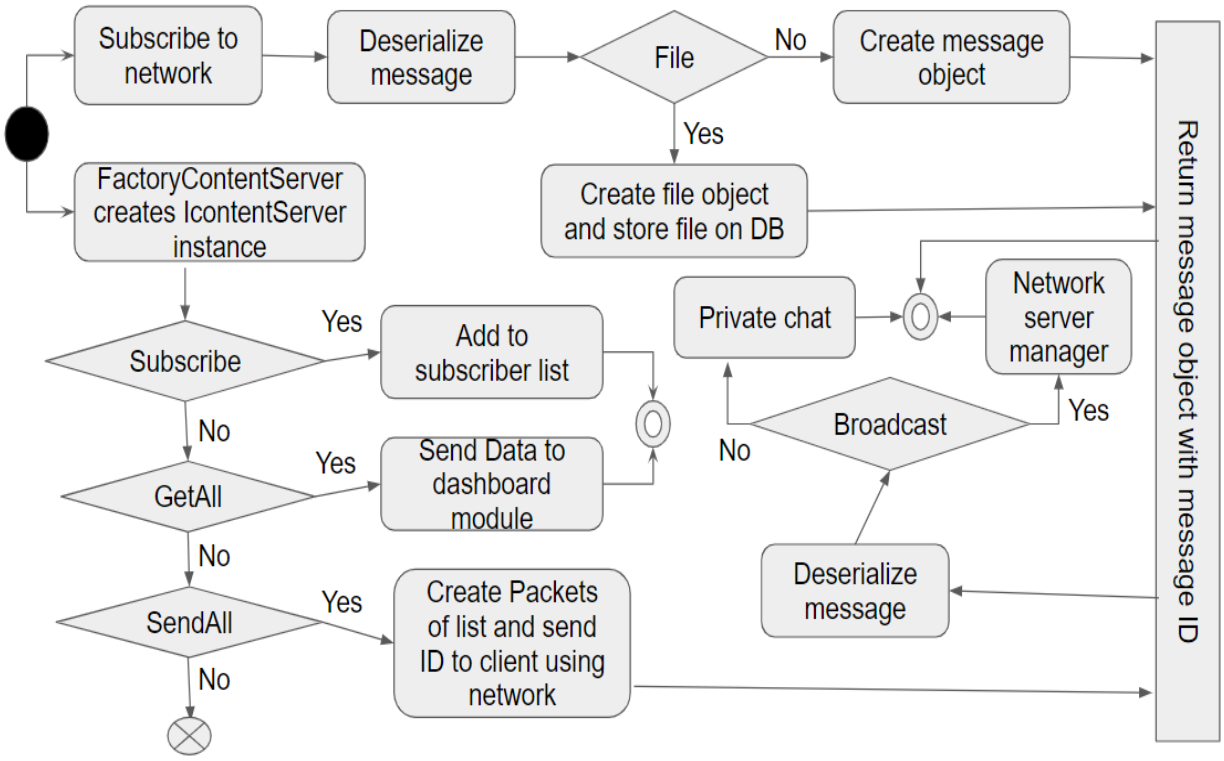


Activity diagram for Content Client component while sending messages



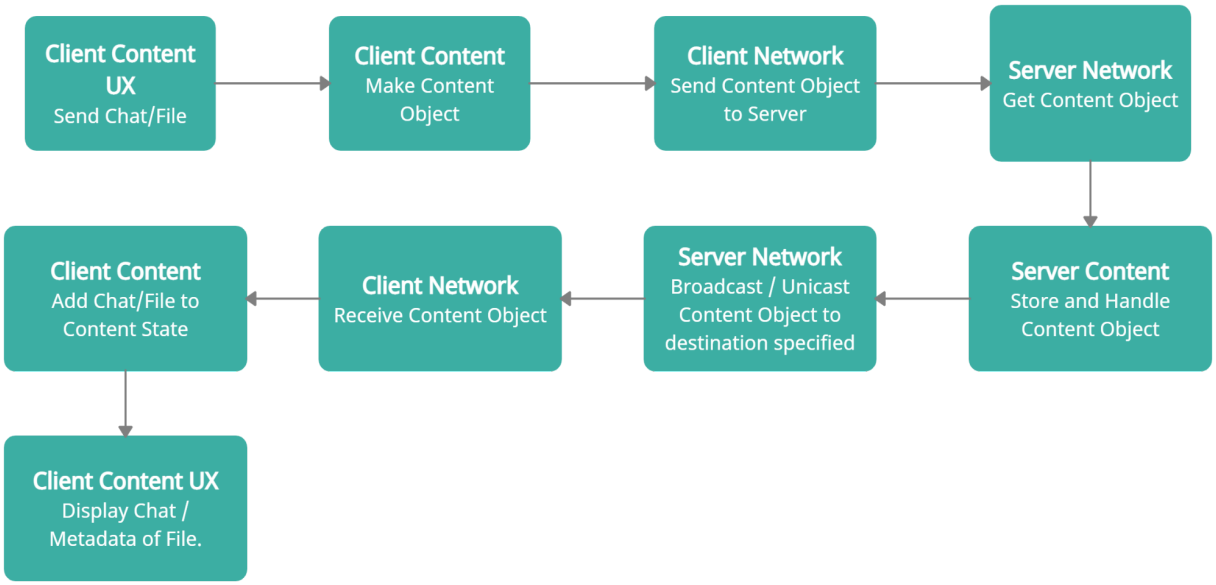
Activity diagram for Content Client component while receiving messages

- **Content Server(By Anurag Jha)**:



Activity diagram for Content Server component

- UX:



Activity diagram of UX

Analysis

We are separating the content module into two parts → Client and Server. This way reduce the complexity and dependency in our module making it easier to make changes on both servers. The Chat system will play a very crucial part during the lab session. As we offer multiple features to communicate, share and receive chats messages and files, it will widely be used the users. Our singleton factory design helps reduce redundancy and offer the following advantages:

- Easier to hide dependencies and maintain.
- Interfaces can be implemented.
- Single Point Access.

We also use publishe subsccriber pattern for sending and receiving messages because:

- Publisher and subscriber do not know each other. Hence reduces complexity,
- Easy to maintain as changes on one side have minimal impact on the other side.

Summary and Conclusion

- A Client and Server side of Content instance will be created at the beginning.
- When a user joins, the ContentListener's OnAllMessage function will be responsible for retrieving the chats and files stored in the content server.
- The users communicate via the Client side.
- The messages and actions are published by the Client side via the network.
- The Server side subscribes and listens to the network and stores the chat and files in it.
- Based on the action request on the View, the ContentViewModel carries on the appropriate action and the changes are reflected on the view
- The chat window will be located on the right-hand side of the application screen. At the bottom of this window there will be text box along with the send button where the users can type in their message. The sent messages appear on the right side of the chat box, while the received messages as well as the username appear on the left side. On hovering over the message, the following options will appear → edit, reply, delete and star. Edit and delete are only available on the messages sent. If a user decides to edit the message, the old message will be overwritten. If a user decides to delete a message, the message will be replaced with "Message Deleted". On replying to a message, the original message will appear above the replied message in the same chat bubble to provide context. The star function will allow you the user to mark the message as important. When a file type message appears, the filename and size are shown, and the user is given the choice to download the file. The user is prompted for the download path to save the file in case they decide to download it.