Criterion C

Techniques Used:

- Encapsulation
- Encryption/Decryption – Algorithmic Thinking
- Generating Invite Codes/Conversation Keys – Algorithmic Thinking
- Authentication
- Polling

Encapsulation

The principle of Encapsulation was used in this web application to allow for better data management and data security. The application was split into two main section (Client and Server) see Image 1. Client is represented by the WebContent folder and Server section is represented by the Java Resources section. Each section has its own subfolders which again are properly structured.

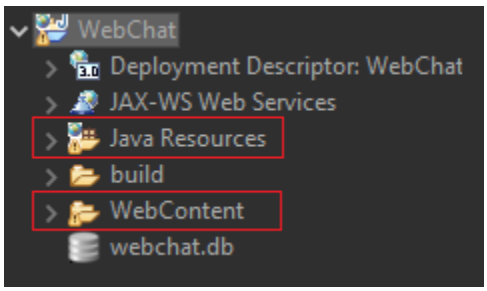

Image 1: Main Structure of Application

The client side (see Image 2), has a folder *"web"*, where all pages of the web application are located, together with scripts, styles and elements that are used on the html pages. The server side of the application (see Image 3), has two folders (*"src"*, which represents the location of the source code and *"Libraries"* which holds libraries that are needed to construct a web application with Java. The *"src"* file is split into 4 primary packages (Api, Business Layer, Data Layer and Model). The model package holds entities that are being used by the application, such as Account, Conversation, (certain types of Request and Responses from the Client to the Server and vice versa) etc. The Api class, receives requests from the client side of the application and outputs responses from the server side based on the request received. The Business layer is responsible for all

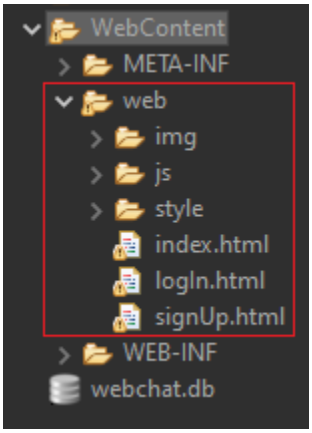

Image 2: Hierarchical View of *"WebContent"* folder

logical events in the web application. The classes hold detailed algorithms that are used for various operations in the application. Finally, the data layer, is responsible for operations with the tables in the database. Encapsulation was necessary to maintain an organized environment while working, which boosted efficiency in implementing features and understanding of how the application works as a whole.

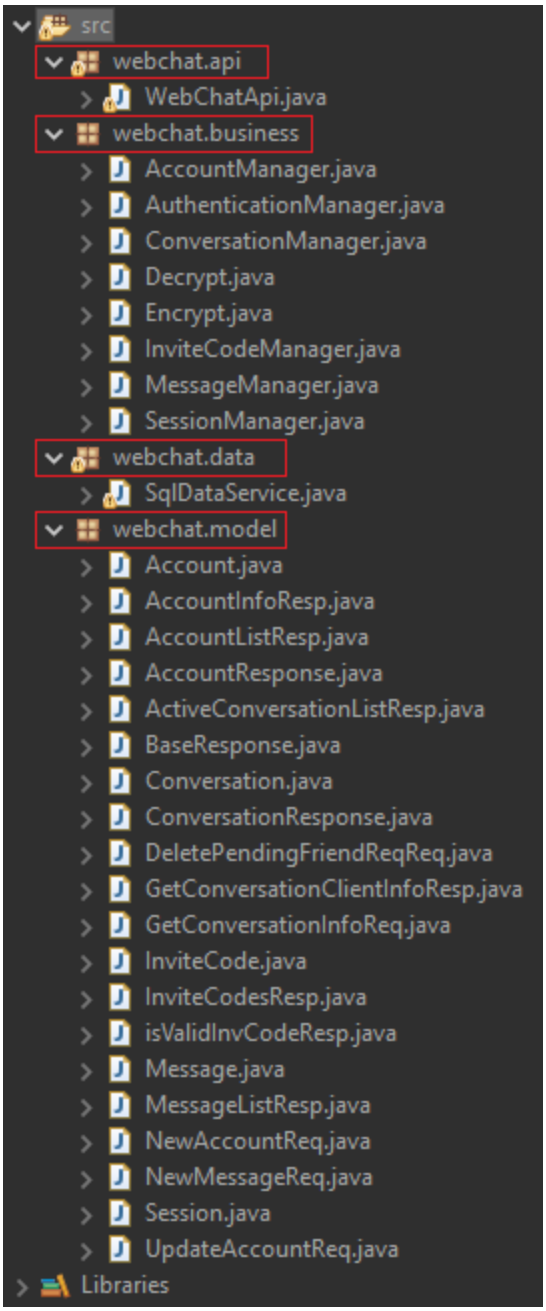

Image 3: Hierarchical View of Source Code Package *"src"*

Personal Code: jfg073

Encryption/Decryption

Every major messaging application has a form of encryption to protect the user's data from being read by possible intruders. Since this chat application was from the beginning meant to be private and anonymous as per the request of the client, encryption and decryption of messages were a must have in the development of this app. The classes *"Encryption"* and *"Decryption"* in the Business package handle the encryption and decryption of messages based on the key of the conversation the messages are part from (see Image 4 for *"Encryption"* and *"Decryption"* classes).

```java
package webchat.business;

public class Encrypt {

    public Encrypt() {
    }

    public String encryptText(String key, String text) {

        String encryptedText = "";

        String characters = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ ";

        int keyLength = key.length();
        int i = 0;

        for (int index = 0; index < text.length(); index++) {

            if(i >= keyLength) {
                i = 0;
            }

            char textChar = text.charAt(index);
            int charPos = characters.indexOf(textChar);

            char encryptedChar;

            if(charPos != -1) {

                char keyChar = key.charAt(i);
                int posAddChar = characters.indexOf(keyChar);
                int newCharPos = charPos + posAddChar;

                if(newCharPos > 62) {

                    newCharPos = newCharPos - characters.length();

                }

                encryptedChar = characters.charAt(newCharPos);

            } else {

                encryptedChar = textChar;

            }

            i++;

            encryptedText = encryptedText + encryptedChar;

        }

        return encryptedText;
    }

}
```

```java
package webchat.business;

public class Decrypt {

    public Decrypt() {
    }

    public String decryptText(String key, String text) {

        String decryptedText = "";

        String characters = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ ";

        int keyLength = key.length();
        int i = 0;

        for (int index = 0; index < text.length(); index++) {

            if(i >= keyLength) {
                i = 0;
            }

            char textChar = text.charAt(index);
            int charPos = characters.indexOf(textChar);

            char decryptedChar;

            if(charPos != -1) {

                char keyChar = key.charAt(i);
                int posSubstractChar = characters.indexOf(keyChar);
                int newCharPos = charPos - posSubstractChar;

                if(newCharPos < 0) {

                    newCharPos = characters.length() + newCharPos;

                }

                decryptedChar = characters.charAt(newCharPos);

            } else {

                decryptedChar = textChar;

            }

            i++;

            decryptedText = decryptedText + decryptedChar;

        }

        return decryptedText;
    }

}
```

Image 4: *"Encryption"* and *"Decryption"* with the encryptText and decryptText method

The methods present in these classes are used when handling messages in the *"MessageManager"* class in the Business package to encrypt the message when it is sent and decrypt a message when the user requests the message of a certain conversation. The methods themselves work as a self-developed Vigenère cipher that occurs between the users, based on a key generated when the conversation was created (see Image 5 for method in class *"Conversation"* (entity) in Model package). The algorithm of generating key's will be developed more in the "Generating Invite Codes/Conversation keys" section. The Vigenère cipher takes the characters of the conversation key and switches the characters of a message based on the position of the character in the conversation key, however, there are certain exceptions that have to be accounted for when using a Vigenère cipher. If the position the letter is supposed to switch to, is larger/smaller (depending whether the program is encrypting or decrypting) than the final/initial position in the alphabet, the Vigenère cipher continues to switch the remaining positions from the beginning/ending of the alphabet. To account for both numbers, lowercase and uppercase letters and spaces, I created a custom string of characters made up of all digits, all possible types of letters and the space character, and applied the exception when certain conditions were met. Line 36 in both *"Encryption"* and *"Decryption"* classes, represents this condition. The reason the position of the new character represented by the variable *"newCharPos"* is compared to 62 (the largest position of the custom string when encrypting) is because the switching of letter occurs from left to right and is compared to 0 (the smallest position of the custom string when decrypting) is because the switching of the letter occurs from right to left.

Generating Invite Codes/Conversation keys

The algorithm implemented in generating Invite Codes and Conversation keys, creates a string of characters based on the Random Java class. Depending for which situation the application is generating a code, it is of different lengths (5 characters for invite codes and 7 characters for conversation keys for increased security, see Image 5 and 6 for two instances where the method is applied).

```java
package webchat.model;

import java.util.Random;

public class Conversation {

    private int id;
    private int id1;
    private int id2;
    private long lastUpdate;
    private String conversationKey;

    public Conversation() {
        this.id = 0;
        this.id1 = 0;
        this.id2 = 0;
        this.lastUpdate = 0;
        this.conversationKey = "";
    }

    public Conversation(int id, int id1, int id2, long lastUpdate, String conversationKey) {
        this.id = id;
        this.id1 = id1;
        this.id2 = id2;
        this.lastUpdate = lastUpdate;
        this.conversationKey = conversationKey;
    }

    public void generateConversationKey() {

        String code = "";
        String characters = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
        Random rand = new Random();
        int random = 0;
        for(int i = 0 ; i < 7; i++) {
            random = rand.nextInt(62);
            code = code + characters.charAt(random);
        }

        this.conversationKey = code;

    }
```

Image 5: Generate code method in *"Conversation"* class (entity) in Model package

```java
package webchat.model;

import java.util.Random;

public class InviteCode {

    private int id;
    private int id_a;
    private String code;

    public InviteCode() {

        this.id_a = -1;
        this.code = "";
    }

    public void generateCode() {
        String code = "";
        String characters = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
        Random rand = new Random();
        int random = 0;
        for(int index = 0; index < 5; index++) {
            random = rand.nextInt(62);
            code = code + characters.charAt(random);
        }
        this.code = code;
    }
```

Image 6: Generate code method in *"InviteCode"* class (entity) in Model package

The algorithm itself works as follows. With the help of the Random Java class, random integers are generated between the values (0 and 62, excluding 62). The random integer generated represents the position of a character in a custom string of letters, that contains all digits and all possible types of letters. The code slowly generates itself through iteration by adding the corresponding character.

This algorithm was necessary for the application for two main reasons:

1. To generate invite codes in order to apply these when a user may try to create an account. This fulfills the client's desire to make this into a private application.
2. To create conversations keys that are used for encrypting and decrypting of messages, thus increase the security of the application and the privacy of users.

Authentication

This technique is necessary to secure an application. It encapsulates (the Logging in event, the use of Sessions and the use of Cookies). Whenever a user accesses the application, the server verifies its identity, based on the cookies the client holds. Cookies are certain files from different websites that are held by a browser. These may be used to hold important information that may be accessed by certain websites to enable some processes. My application requires users to log in, to access the rest of the features of the application. Whenever a user logs in, a request to access the application is sent to the server, which contains credentials, an Account entity that is verified by the server. After the server determines the legitimacy of the information received, a Session (entity that holds a record of whether a user is logged in the application or not and for how long) is created and the id of that session is saved into a cookie and sent to the client through a server response. Since for every action that a user takes in the application the server verifies if the user is authenticated (has a valid session that has not expired), upon receiving requests, the server checks for the cookie on the user's computer that holds the session id to be verified. Log out is also a required method for certain users that want to leave the computer unattended while not closing the application to be able to maintain their privacy and keep themselves save from people impersonating them. This involves the deletion of cookies present on the user's computer and the update of the user's session in the data base to set it at expired. The algorithms previously described are detailed in Image 7 of the *"AuthenticationManager"* class in Busines package. Method used to log in is underlined with red, method used to check if a user is authenticated is underlined in orange and method that logs out a user is underlined in light blue.

Personal Code: jfg073

```java
package webchat.business;

import java.util.Date;

public class AuthenticationManager {

    public AccountResponse logIn(Account accountToVerify, HttpServletResponse httpResponse, HttpServletRequest httpRequest) throws Exception {

        AccountResponse resp = new AccountResponse();

        try {

            AccountManager am = new AccountManager();
            Account databaseAccount = am.getAccountByID(accountToVerify.getId());
            String dataBasePass = databaseAccount.getPassword();
            String dataBaseUser = databaseAccount.getUsername();
            if(!dataBasePass.contentEquals(accountToVerify.getPassword()) || !dataBaseUser.contentEquals(accountToVerify.getUsername())) {

                throw new Exception ("Incorrect Username or Password!");

            }
            Session s = new Session();
            SessionManager sm = new SessionManager();
            String sessionId = sm.insertSession(s);
            Cookie c = new Cookie("sessionId", sessionId);
            c.setMaxAge(1000*60*30);
            Cookie cAccID = new Cookie("accountID", String.valueOf(accountToVerify.getId()));
            cAccID.setMaxAge(1000*60*30);
            httpResponse.addCookie(c);
            httpResponse.addCookie(cAccID);

        } catch (Exception e) {

            resp.setErrorByException(e);
        }

        return resp;

    }

    public HttpServletResponse logOut(HttpServletRequest httpRequest, HttpServletResponse httpResponse) throws Exception {

        Cookie [] cookieList = httpRequest.getCookies();

        String cookieSessionID = "";

        for(Cookie cookie  : cookieList) {

            if(cookie.getName().contentEquals("sessionId")) {

                cookieSessionID = cookie.getValue();

            }

            cookie.setMaxAge(0);
            cookie.setValue(null);
            httpResponse.addCookie(cookie);

        }

        SqlDataService dataService = new SqlDataService();
        Session s = dataService.getSessionById(cookieSessionID);
        Date dNow = new Date();
        long sExpirationDate = dNow.getTime();
        s.setExpirationDate(sExpirationDate);
        dataService.updateSession(s);

        return httpResponse;

    }
```

```
83      public boolean isAuth(HttpServletRequest httpRequest) throws Exception {
84
85          boolean isAuth = true;
86
87          Date d = new Date();
88
89          Cookie [] cookieList = httpRequest.getCookies();
90
91          if(cookieList == null) {
92
93              throw new Exception ("No Cookies");
94
95          }
96
97          String cookieSessionID = "";
98
99          for(Cookie cookie : cookieList) {
100
101             if(cookie.getName().contentEquals("sessionId")) {
102
103                 cookieSessionID = cookie.getValue();
104
105             }
106
107         }
108
109         SqlDataService dataService = new SqlDataService();
110         Session s = dataService.getSessionById(cookieSessionID);
111         Date sExpirationDate = new Date(s.getExpirationDate());
112
113
114         if(d.after(sExpirationDate)) {
115
116             isAuth = false;
117
118         }
119
120         return isAuth;
121
122     }
123
124 }
```

Image 7: "*AuthenticationManager"* class from Business package, with *"logIn"*, *"logOut"* and *"isAuth"* methods

Polling

Polling in Computer Science and more specifically where I use this process in my web application, it refers to the client continuously creating server requests to check for new messages and if the server determines that new messages exist, it decrypts and sends them to the client side for the user to see. This is achieved through a JQuery function *"setInterval"* which executes a function (*"doPoll"* (personal developed JavaScript function)) repeatedly every (desired time /ms) (see Image 8 for the *"doPoll"* function). The way this algorithm works is as following. The client sends a request to the server to check for new messages, the request including: the conversation id and when was the conversation last updated. This information is passed on the server side. From client cookies, the application is able to obtain the conversation id and check when was this conversation last updated. If the conversation was last updated on the server at a later date (see red lines in Image 9 for the name of the function in *"MessageManager"* class in Business model that handles this operation) than it was last updated on the client, the application gets the latest messages from the database and sends them back to the user, where they are added to the conversation container (see Image 9 for described algorithm).

```
1605            function doPoll() {
1606
1607                if (!process) {
1608
1609                    refreshMessages();
1610                    var scrollPos = $(".messages").scrollTop();
1611                    const messagesDivHeight = $(".messages").innerHeight();
1612                    var value = $(".messages")[0].scrollHeight;
1613                    if(scrollPos + messagesDivHeight >= value) {
1614
1615                        $(".messages").animate({ scrollTop: $(".messages")[0].scrollHeight}, 1000);
1616
1617                    }
1618
1619                }
1620
1621            }
```

Image 8: *"doPoll"* function applicable specifically to my application

Personal Code: jfg073

```java
    /**
     * API that returns the information needed for a conversation page
     * It receives two account iDs
     * @return A GetConversationClientInfoResponse object (inheriting BaseRespponse class) that contains error code and description (from the BaseResponse class)
     * and useful information for client side
     */
    @POST
    @Path("/getConversationInfo")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public BaseResponse getConversationInfo(GetConversationInfoReq req) {

        GetConversationClientInfoResp resp = new GetConversationClientInfoResp();

        try {

            AuthenticationManager authM = new AuthenticationManager();

            if(authM.isAuth(httpRequest)) {

                int convId = req.getIdConv();

                boolean flag = false;
                String cookieName = "conversationKeyOfId_" + convId;
                String conversationKey = "";

                Cookie [] cookieList = httpRequest.getCookies();

                int cookieAccountID = -1;

                for(Cookie cookie : cookieList) {

                    if(cookie.getName().contentEquals("accountID")) {

                        cookieAccountID = Integer.valueOf(cookie.getValue());

                    }

                    if(cookie.getName().contentEquals(cookieName)) {

                        flag = true;
                        conversationKey = cookie.getValue();

                    }

                }

                ConversationManager cm = new ConversationManager();
                Conversation c = cm.getConversationByID(convId);

                int id1 = c.getId1();
                int id2 = c.getId2();

                if(cookieAccountID != id1 && cookieAccountID != id2) {

                    throw new Exception ("Access Denied!");

                }

                if(!flag) {

                    conversationKey = c.getConversationKey();
                    Cookie cConvKey = new Cookie(cookieName, conversationKey);
                    cConvKey.setMaxAge(-1);
                    httpResponse.addCookie(cConvKey);

                }
```

```
1137                long lastUpdateClient = req.getLastUpdate();
1138                int offSet = req.getOffSet();
1139                int limit = 11;
1140                boolean isPolling = req.getIsPolling();
1141
1142                AccountManager am = new AccountManager();
1143                MessageManager mm = new MessageManager();
1144
1145                Account acc = new Account();
1146
1147                if(cookieAccountID != id1) {
1148
1149                    acc = am.getAccountByID(id1);
1150
1151                } else {
1152
1153                    acc = am.getAccountByID(id2);
1154
1155                }
1156
1157                c.setConversationKey("");
1158                c.setId1(-1);
1159                c.setId2(-1);
1160
1161                acc.setPassword("");
1162
1163                MessageListResp messagesResp = new MessageListResp();
1164
1165                if(!isPolling) {
1166
1167                    messagesResp = mm.getLatestMessages(convId, id1, id2, limit, offSet, conversationKey);
1168
1169                } else {
1170
1171                    messagesResp = mm.getLatestMessagesPoll(convId, id1, id2, limit, offSet, lastUpdateClient, conversationKey);
1172
1173                }
1174
1175                resp.setMessagesResp(messagesResp);
1176                resp.setAccount(acc);
1177                resp.setConv(c);
1178
1179
1180            } else {
1181
1182                throw new Exception ("Session has Expired!");
1183
1184            }
1185
1186        } catch (Exception e) {
1187
1188            resp.setErrorByException(e);
1189
1190        }
1191
1192        return resp;
1193
1194    }
```

Image 9: *"getConversationInfo"* method in the *"Api"* class in Api package that handles messages of a conversation, additionally to other information

Word Count: 1227