Assignment Cover Letter

(Individual Work)

Student Information:

Surname	Given Names	Student ID Number
Wijadi	Jason Jeremy	2440077301

Course Code	**: COMP6699**	**Course Name**	**: Object Oriented Programming**
Class	**: L2BC**	**Name of Lecturer(s)**	**: Jude Joseph Lamug Martinez**
Major	**: Computer Science**		
Title of Assignment	**: Crude Email**		

Type of Assignment : Final Project

Submission Pattern

Due Date : 22/06/2021 **Submission Date : 22/06/2021**

The assignment should meet the below requirements.

1. **Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.**
2. **Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.**
3. **The above information is complete and legible.**
4. **Compiled pages are firmly stapled.**
5. **Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.**

Plagiarism/Cheating

BiNus International seriously regards all forms of plagiarism, cheating and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

Declaration of Originality

By signing this assignment, I understand, accept and consent to BiNus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student:

Jason Jeremy Wijadi

1. Table of Contents

2. Project Specification

The objective of this program is to create an interface between the user and their email inbox. The program would be able to accept user input to login to their email inbox depending on their email provider. The user would then be able to interact with their email. The information that will be provided immediately would be the sender, subject, content, along with the date and time of when the email was sent. Users would also be able to download and view attachments. In addition to that, users can also compose messages with / without attachments to be sent to other users.

1. Program **input**:
    a. Mouse clicks (Selecting messages, downloading / choosing attachments)
    b. User keyboard input (Logging in, composing messages)
    c. Files to be sent as attachments
2. Program **output**:
    a. Email information (Sender, subject, content, date and time)
    b. Files from attachments

Third-party dependencies / tools used:

1. JavaFX
    - Program Main GUI
2. SceneBuilder
    - JavaFX visual layout tool
3. JakartaMail
    - API for Java email interface
4. Maven
    - Build automation tool
    - Exporting of shaded JAR file
5. IntelliJ IDEA
    - Project IDE
    - UML diagram creator

3. Solution Design

The GUI for the program centers around a single window for logging in and receiving email. Each different interface will close the previous one and open a new window (special case for the sending window where it is its own separate window).
The list of the interfaces are as follows:

1. Landing Window
2. Login Gmail Window
3. Login Outlook Window
4. Main Window
5. Send Window

Landing Window

The user starts the program on the landing window. There are two buttons on this window: Gmail, and Outlook (Experimental). Each of these buttons will lead the user to the linked interface that the button is associated with. An explanation on why the Outlook part of the project is still experimental will be explained soon.

Login Windows (Gmail and Outlook)

The two login windows represent the two providers that the program supports, Gmail and Outlook. They both have two input fields for the email address and the associated password along with a button for logging in. Once the button is pressed, it will be disabled until the authentication progress finishes. An error will show up if something goes wrong, otherwise the program will continue to the main window. A button to go back to the landing page is also present in case the user wants to switch providers.

Main Window

The main window hosts the bulk of the application. A view of the folders that the email account holds will be displayed on the left side while its contents will be displayed in the middle. The sender, subject, content snippet, along with date and time are readily available for the user to see. They will also be able to determine if the specific email has been read or not and if it holds an attachment or not. Once a

message / email is selected, its content will be displayed on the right side along with more information on its properties. If there are attachments, buttons will appear for the user to click on and download the attachments to their device. They can also click on the button again to view the file externally.

Send Window

The send window will be available to the user in a menu bar on the main window. Once clicked, the window will appear. Input fields for the recipient(s) and subject of the email will be present along with a HTML editor for the contents of the email. Attachment sending is also supported through a button which will let the user pick any file from their computer to send.

Program Notes

In certain cases, the program will launch but no components will be displayed. This may be due to the graphics accelerator that Java chooses to use when running the program. To circumvent this, run the program with additional arguments / VM options:

```
-Dprism.order=sw
```

Java version for this project = 15.0.2

Demo video link: youtu.be/JhZT52WVFuQ

Installation instructions are on the GitHub repository at github.com/digaji/CrudeEmail

3.1 UML Diagram

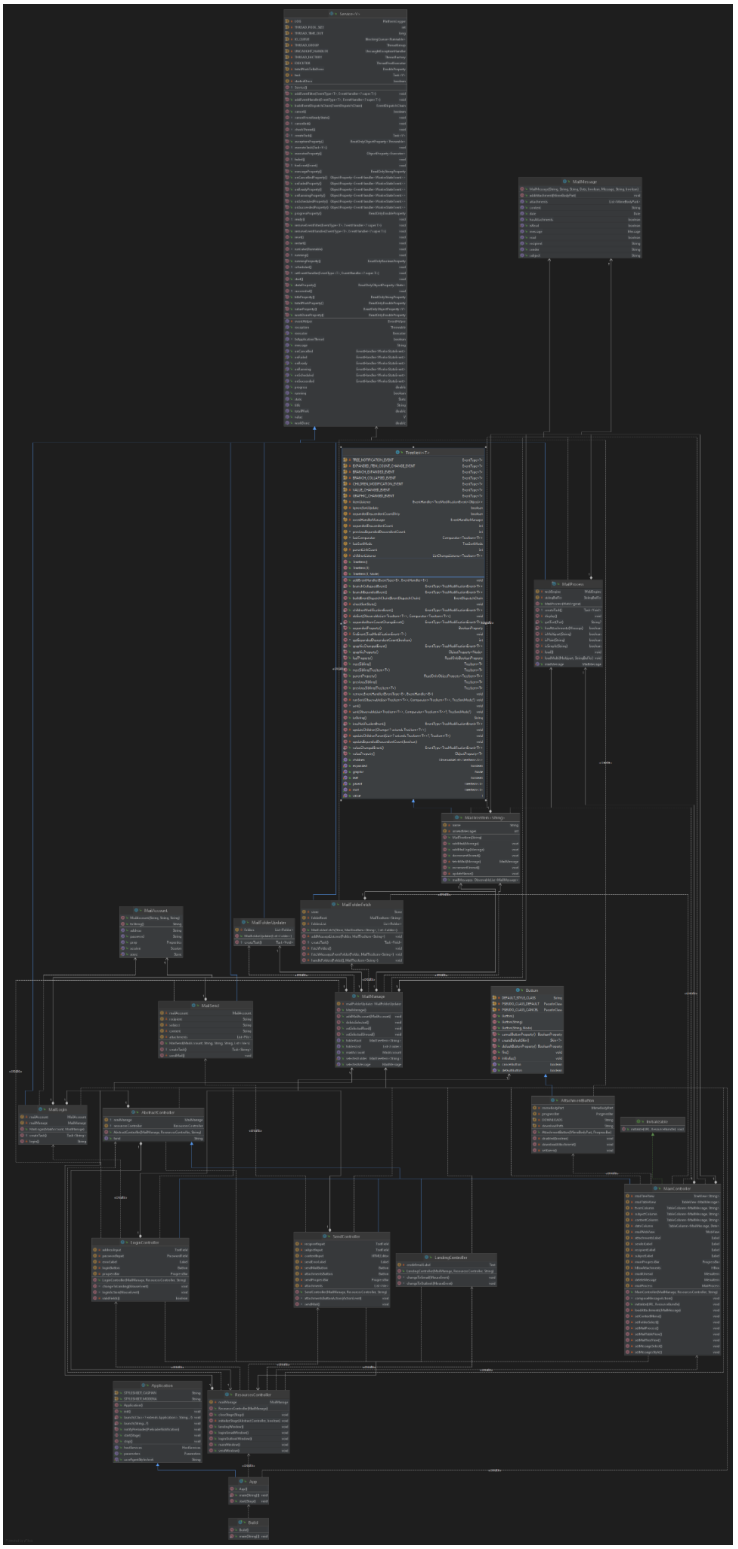


Fig. 1: Crude Email whole UML diagram

UML also available in /crudeemail-uml.png or /crudeemail.uml

The very long parts are derived classes / interfaces from JavaFX.

3.2 File Structure

Below is the final file structure for Crude Email:

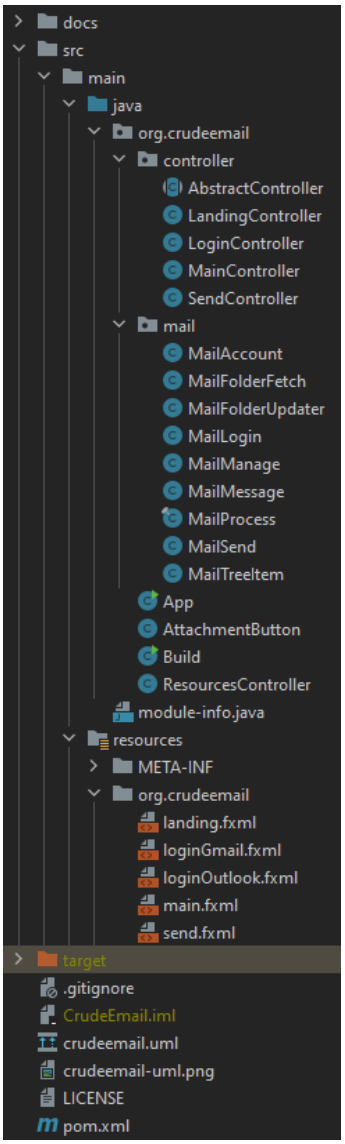


Fig. 2: Crude Email file structure

All Java files are stored in the java folder while the resources folder holds all things related to JavaFX fxml. pom.xml is used to determine Maven what dependencies need to be used for the project.

The docs folder holds the JavaDoc HTML files created based on the comments made in the code. This can be viewed on github-pages in the repository.

3.3 How It Works / App Flow

3.3.1 App Class and ResourceController

It all starts from the App class that extends from JavaFX Application.

```java
public class App extends Application {

    @Override
    public void start(Stage stage) throws Exception {
        ResourcesController resourcesController = new ResourcesController(new MailManage());
        resourcesController.landingWindow();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Fig. 3: App snippet

The main controller that handles all the windows is instantiated here and the landing window is called through a method.

```java
// Call in stages
public void landingWindow() {
    System.out.println("Landing window called\n");

    AbstractController controller = new LandingController(mailManage, resourcesController: this, fxml: "landing.fxml");
    initializeStage(controller, resizable: false);
}

public void loginGmailWindow() {
    System.out.println("Login Gmail window called\n");

    AbstractController controller = new LoginController(mailManage, resourcesController: this, fxml: "loginGmail.fxml");
    initializeStage(controller, resizable: false);
}

public void loginOutlookWindow() {
    System.out.println("Login Outlook window called\n");

    AbstractController controller = new LoginController(mailManage, resourcesController: this, fxml: "loginOutlook.fxml");
    initializeStage(controller, resizable: false);
}

public void sendWindow() {
    System.out.println("Send window called\n");

    AbstractController controller = new SendController(mailManage, resourcesController: this, fxml: "send.fxml");
    initializeStage(controller, resizable: false);
}

public void mainWindow() {
    System.out.println("Main window called\n");

    AbstractController controller = new MainController(mailManage, resourcesController: this, fxml: "main.fxml");
    initializeStage(controller, resizable: true);
}
```

Fig. 4: ResourcesController window methods

Each window has its own controller called with some things passed in as arguments in order to make them work (explanation for this coming soon). Then the initialize stage method is called.

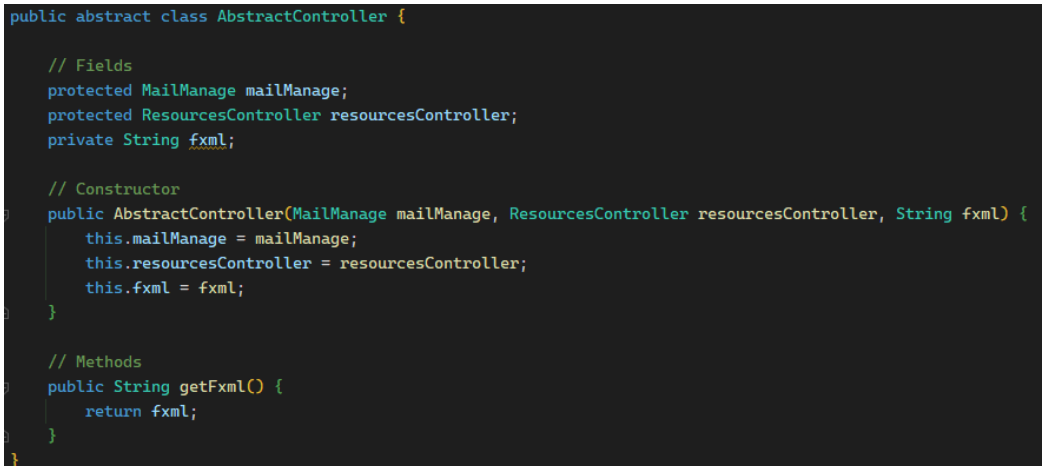
```java
// Stage handling
private void initializeStage(AbstractController abstractController, boolean resizable) {
    FXMLLoader fxmlLoader = new FXMLLoader(getClass().getResource(abstractController.getFxml()));
    fxmlLoader.setController(abstractController);

    Parent parent;

    try {
        parent = fxmlLoader.load();
    } catch (IOException e) {
        e.printStackTrace();
        return;
    }

    Scene scene = new Scene(parent);
    Stage stage = new Stage();
    stage.setScene(scene);
    stage.setTitle("Crude Email");
    stage.initStyle(StageStyle.UNIFIED);

    if (!resizable) {
        stage.setResizable(false);
    }

    // Special minimum width and height for main.fxml
    if (abstractController.getFxml().equals("main.fxml")) {
        stage.setMinWidth(1500);
        stage.setMinHeight(720);
    }

    stage.show();
}

public void closeStage(Stage stage) {
    stage.close();
}
```

Fig. 5: ResourcesController window handling methods

The fxml is loaded while also setting the specified controller to it. This was done instead of just assigning the controller in the fxml in order to pass through those other things in the argument. In the case of main.fxml, a special minimum size is specified so that the user doesn't resize the window to too small of a size.

3.3.2 AbstractController

The AbstractController is what all the other controllers derive from.



```java
public abstract class AbstractController {

    // Fields
    protected MailManage mailManage;
    protected ResourcesController resourcesController;
    private String fxml;

    // Constructor
    public AbstractController(MailManage mailManage, ResourcesController resourcesController, String fxml) {
        this.mailManage = mailManage;
        this.resourcesController = resourcesController;
        this.fxml = fxml;
    }

    // Methods
    public String getFxml() {
        return fxml;
    }
}
```

Fig. 6: AbstractController entirety

All the other controllers derive from this class in order to be able to pass through a MailManage class object to it. Through this MailManage class object, the email account is accessed and managed along with some high level things before the actual email is interacted with. All the windows will get this object in order to allow for constant interaction with the same email account later on.

3.3.3 LandingController

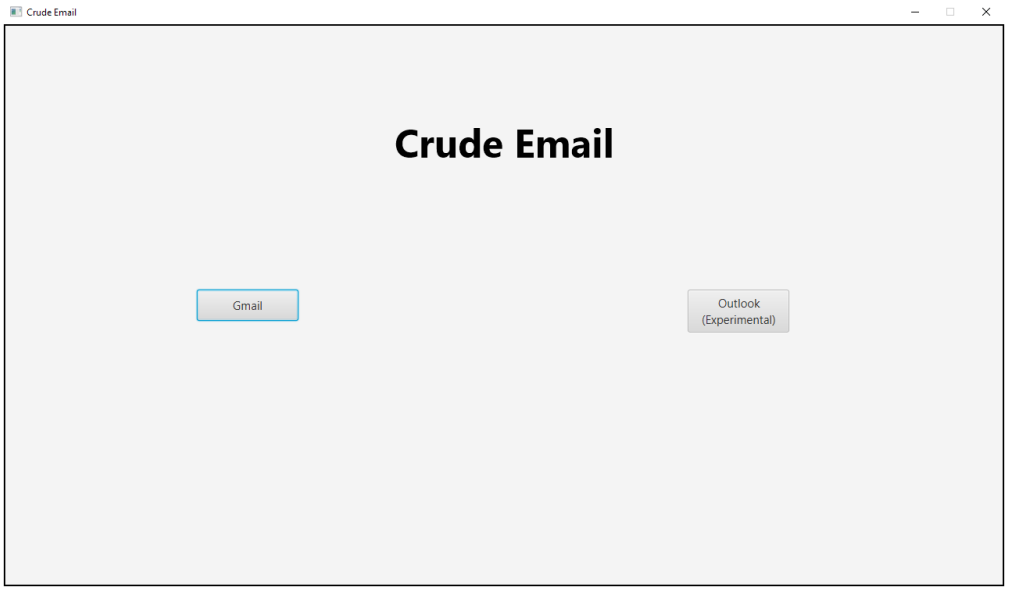


Fig. 7: Landing Window

The LandingController interfaces the changes to the loginGmail or Outlook window.

3.3.4 LoginController, MailAccount, and MailLogin

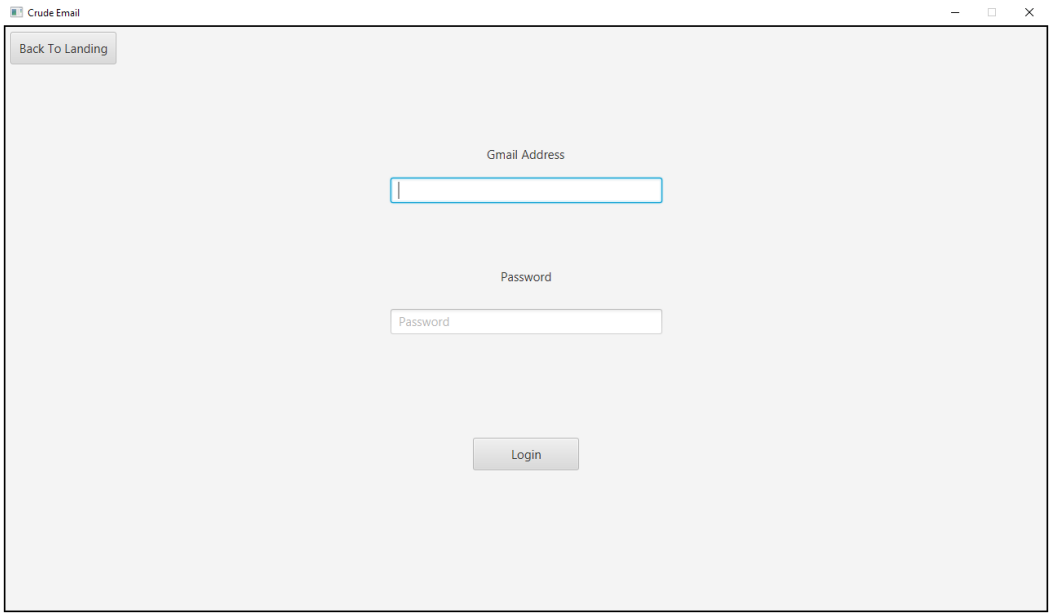


Fig. 8: LoginGmail Window

The LoginController manages what happens in both LoginGmail and LoginOutlook.

```
@FXML
void loginAction(MouseEvent mouseEvent) {
    if (validFields()) {
        String provider;
        // Set provider depending on window being opened

        if (getFxml().equals("loginGmail.fxml")) {
            provider = "gmail";
        } else {
            provider = "outlook";
        }

        MailAccount mailAccount = new MailAccount(provider, addressInput.getText(), passwordInput.getText());
        MailLogin login = new MailLogin(mailAccount, mailManage);

        loginButton.setDisable(true);

        // Start and execute login JavaFX service for multithreading
        login.start();
        login.setOnSucceeded(event -> {

            String result = login.getValue();

            switch (result) {
                case "SUCCESS":
                    System.out.println("Logged in to: " + mailAccount + "\n");

                    // Switch to main window
                    resourcesController.mainWindow();
                    Stage currentStage = (Stage) addressInput.getScene().getWindow();
                    resourcesController.closeStage(currentStage);
                    break;
                case "CREDENTIALS FAILED":
                    errorLabel.setText("INVALID CREDENTIALS");
                    break;
                case "UNEXPECTED ERROR":
                    errorLabel.setText("UNEXPECTED ERROR");
                    break;
                case "NETWORK FAILED":
                    errorLabel.setText("NETWORK FAILED");
                    break;
            }

            // Reset fields
            addressInput.setText("");
            passwordInput.setText("");
            loginButton.setDisable(false);
            progressBar.setOpacity(0);
        });
    }
}
```

Fig. 8: LoginController loginAction method

This loginAction method is called when the login button is pressed. Many things happen at once here. First the input fields are checked to make sure that they are all filled. Then a new MailAccount object is created depending on if it's the LoginGmail or LoginOutlook window that the button is in. Next, a login attempt is made through the MailLogin object's createTask() method (called implicitly through the start() method). Once authentication is done, the returned result will determine if the credentials are valid and if the main window will be called or not. Otherwise, an error label will show on the login window and all the text will be cleared. While this is all happening, the login button is disabled and an indeterminate progress bar is shown to indicate that the process is running.

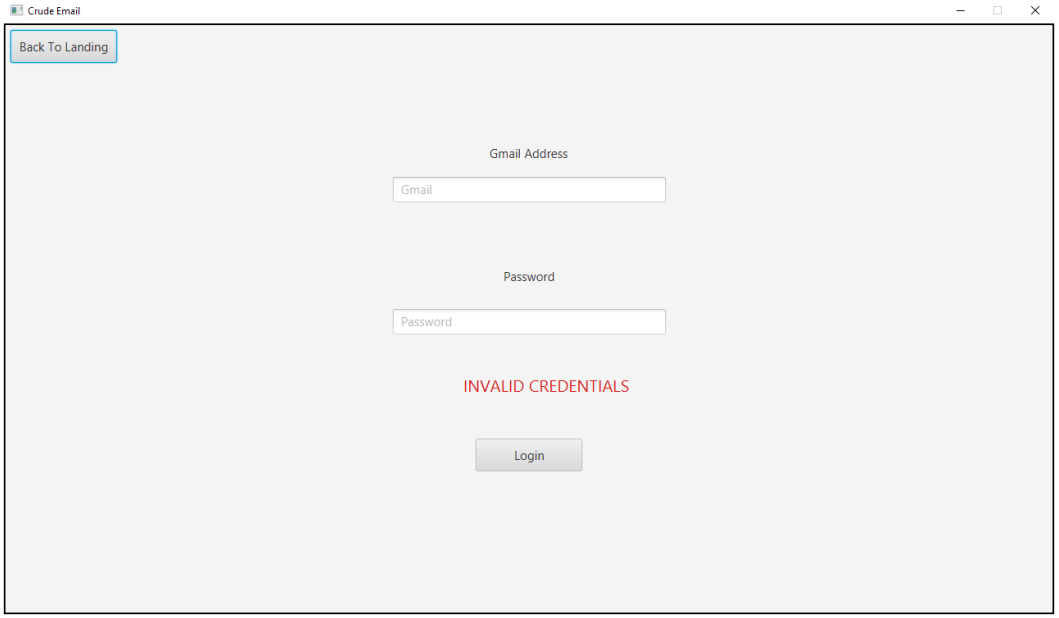


Fig. 9: Example error when invalid credentials are entered

```java
// Fields
private String address;
private String password;
private Properties prop;
private Store store;
private Session session;

// Constructor(s)
public MailAccount(String provider, String address, String password) {
    prop = new Properties();
    if (provider.equals("gmail")) {
        // Gmail settings
        prop.put("mail.smtp.host", "smtp.gmail.com");

        prop.put("mail.imap.host", "imap.gmail.com");
    } else {
        // Outlook settings
        prop.put("mail.smtp.host", "smtp.office365.com");

        prop.put("mail.imap.host", "outlook.office365.com");
    }

    // General settings
    prop.put("mail.smtp.port", "587");
    prop.put("mail.smtp.auth", "true");
    prop.put("mail.imap.port", "993");
    prop.put("mail.imap.ssl.enable", "true");
    prop.put("mail.smtp.starttls.enable", "true");

    this.address = address;
    this.password = password;
}
```

Fig. 10: MailAccount snippet

The MailAccount class holds getters and setters for its fields and also this constructor. It determines the settings for IMAP receiving and SMTP sending configurations for both Gmail and Outlook.

```java
// Methods
private String login() {
    // Authenticate account with Authenticator class object
    Authenticator authenticator = new Authenticator() {
        @Override
        protected PasswordAuthentication getPasswordAuthentication() {
            return new PasswordAuthentication(mailAccount.getAddress(), mailAccount.getPassword());
        }
    };

    try {
        Session session = Session.getInstance(mailAccount.getProp(), authenticator);
        mailAccount.setSession(session);

        // Create Store object to connect to servers
        Store storeObj = session.getStore( protocol: "imap");
        // Connect to store object
        storeObj.connect(mailAccount.getProp().getProperty("mail.imap.host"),
                mailAccount.getAddress(),
                mailAccount.getPassword());

        mailAccount.setStore(storeObj);
        mailManage.addMailAccount(mailAccount);
    } catch (NoSuchProviderException e) {
        e.printStackTrace();
        System.out.println("Network Failed");
        return "NETWORK FAILED";
    } catch (AuthenticationFailedException e) {
        e.printStackTrace();
        System.out.println("Credentials Failed");
        return "CREDENTIALS FAILED";
    } catch (Exception e) {
        e.printStackTrace();
        System.out.println("Unexpected Error");
        return "UNEXPECTED ERROR";
    }

    return "SUCCESS";
}

@Override
protected Task<String> createTask() {
    return new Task<>() {
        @Override
        protected String call() throws Exception {
            return login();
        }
    };
}
```

Fig. 10: MailLogin snippet

MailLogin's login() method is called from the createTask() method when the LoginController calls for the start() method. MailLogin extends the JavaFX class Service in order to allow multithreading. This means that when the login process is being executed, the program won't stop / freeze up for the user.

Many parts of the application utilize the Service class in order for multiple processes to happen concurrently and without interruption to the user.

3.3.5 MainController

The bulk of the application revolves around the MainController and the Main Window.

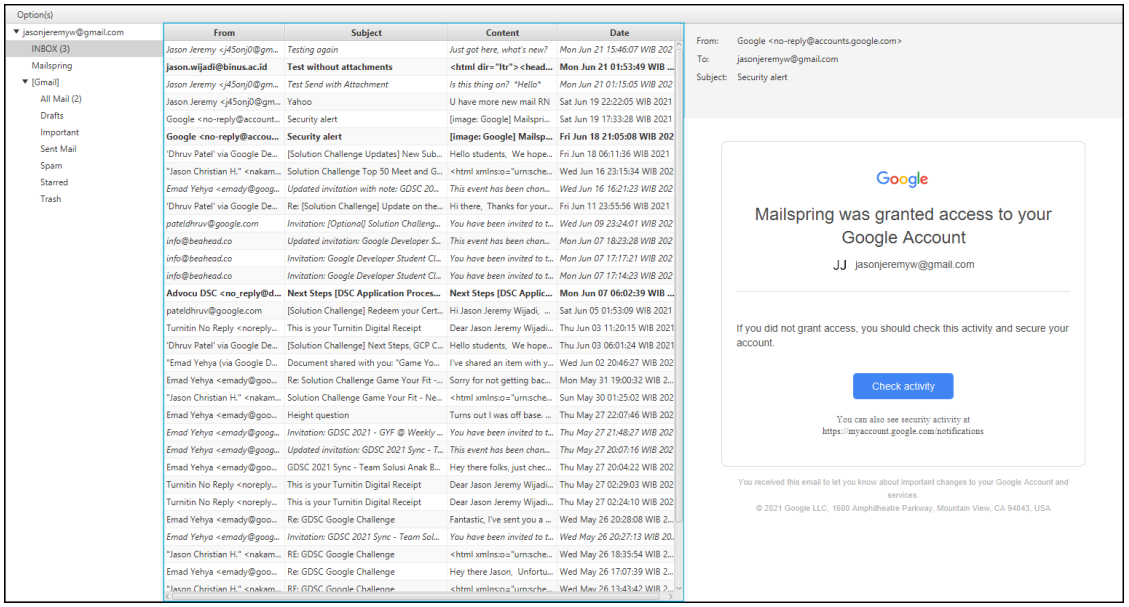


Fig. 11: Main Window sample

```
@Override
public void initialize(URL url, ResourceBundle resourceBundle) {
    // Initialization methods
    setMailTreeView();
    setMailTableView();
    setFolderSelect();
    setMessagesStyle();
    setMailProcess();
    setMessageSelect();
    setContextMenu();
}
```

Fig. 12: MainController initialize method

MainController implements the Initializable interface from JavaFX. This method gets called when the Main Window gets called on. Here, some methods are called in order to set up all the appropriate processes for full functionality in the Main Window.

```java
private void setMailTreeView() {
    // Set the root of the treeView as the first folder
    mailTreeView.setRoot(mailManage.getFolderRoot());
    mailTreeView.setShowRoot(false);
}

private void setMailTableView() {
    // Determine how to populate cells in TableView with cell value factories
    fromColumn.setCellValueFactory(new PropertyValueFactory<>("sender"));
    subjectColumn.setCellValueFactory(new PropertyValueFactory<>("subject"));
    contentColumn.setCellValueFactory(new PropertyValueFactory<>("content"));
    dateColumn.setCellValueFactory(new PropertyValueFactory<>("date"));

    // Add right click context menu
    mailTableView.setContextMenu(new ContextMenu(markUnread, deleteMessage));
}
```

Fig. 13: MainController setMailTreeView and setMailTableView methods

The setMailTreeView and setMailTableView methods set up the left and middle parts of the Main Window. The mailTreeView utilizes a local mailManage object to get the folder root while the mailTableView gets populated with cell value factories.

```java
private void setFolderSelect() {
    // Determine what happens when a specific mail folder is selected, also ignores mouse clicks to anything other than folders
    mailTreeView.setOnMouseClicked(event -> {
        MailTreeItem<String> current = (MailTreeItem<String>) mailTreeView.getSelectionModel().getSelectedItem();

        if (current != null) {
            mailManage.setSelectedFolder(current);
            mailTableView.setItems(current.getMailMessages());
        }
    });
}
```

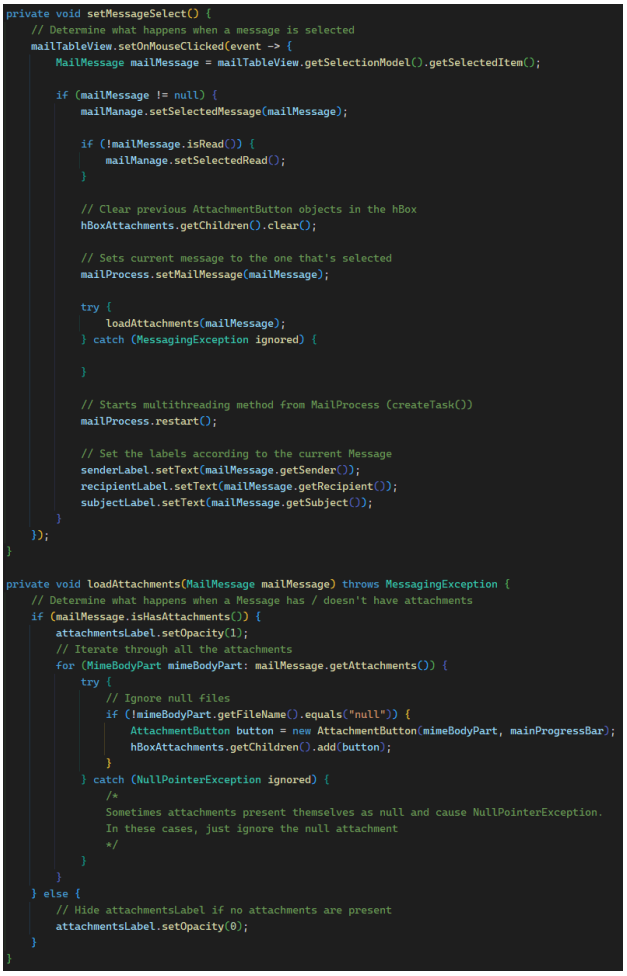
Fig. 14: MainController setFolderSelect method

The setFolderSelect method determines what messages to show when a certain folder is selected. This is to prevent messages from overlapping with each other in the main tableView.

```java
private void setMessagesStyle() {
    // Set the rows with unread messages as bolded and attachments with italic
    mailTableView.setRowFactory(new Callback<>() {
        @Override
        public TableRow<MailMessage> call(TableView<MailMessage> param) {
            // New TableRow object with overridden method
            return new TableRow<>() {
                @Override
                protected void updateItem(MailMessage message, boolean empty) {
                    super.updateItem(message, empty);

                    if (message != null) {
                        if (message.isRead()) {
                            setStyle("");
                        } else {
                            setStyle("-fx-font-weight: bold");
                        }

                        if (message.isHasAttachments()) {
                            setStyle("-fx-font-style: italic");
                        }
                    }
                }
            };
        }
    });
}
```

Fig. 15: MainController setMessagesStyle method

The setMessagesStyle method determines the JavaFX style that the row of that message uses depending on if it's read or not and if it has attachments or not.



```
private void setMessageSelect() {
    // Determine what happens when a message is selected
    mailTableView.setOnMouseClicked(event -> {
        MailMessage mailMessage = mailTableView.getSelectionModel().getSelectedItem();

        if (mailMessage != null) {
            mailManage.setSelectedMessage(mailMessage);

            if (!mailMessage.isRead()) {
                mailManage.setSelectedRead();
            }

            // Clear previous AttachmentButton objects in the hBox
            hBoxAttachments.getChildren().clear();

            // Sets current message to the one that's selected
            mailProcess.setMailMessage(mailMessage);

            try {
                loadAttachments(mailMessage);
            } catch (MessagingException ignored) {

            }

            // Starts multithreading method from MailProcess (createTask())
            mailProcess.restart();

            // Set the labels according to the current Message
            senderLabel.setText(mailMessage.getSender());
            recipientLabel.setText(mailMessage.getRecipient());
            subjectLabel.setText(mailMessage.getSubject());
        }
    });
}

private void loadAttachments(MailMessage mailMessage) throws MessagingException {
    // Determine what happens when a Message has / doesn't have attachments
    if (mailMessage.isHasAttachments()) {
        attachmentsLabel.setOpacity(1);
        // Iterate through all the attachments
        for (MimeBodyPart mimeBodyPart: mailMessage.getAttachments()) {
            try {
                // Ignore null files
                if (!mimeBodyPart.getFileName().equals("null")) {
                    AttachmentButton button = new AttachmentButton(mimeBodyPart, mainProgressBar);
                    hBoxAttachments.getChildren().add(button);
                }
            } catch (NullPointerException ignored) {
                /*
                Sometimes attachments present themselves as null and cause NullPointerException.
                In these cases, just ignore the null attachment
                */
            }
        }
    } else {
        // Hide attachmentsLabel if no attachments are present
        attachmentsLabel.setOpacity(0);
    }
}
```

Fig. 16: MainController setMessageSelect and loadAttachments method

The setMessageSelect method determines what happens when a message is selected. It gets the message that is being selected, sets it as read, changes the message in a local MailProcess object to the current one, loads its attachments, and sets the details for the view on the right. Demo for attachments can be seen in the demo video.

3.3.6 MailManage, MailFolderFetch, and MailFolderUpdater

Now maybe you're wondering how the emails are processed from the MailManage object that gets passed around to the different windows. This is where we get to that.

The basic gist of how Jakarta Mail works is that an account (data from MailAccount) gets authenticated, then an email accessing session can start in which a Store class object connects to the hosting server for the email account (all done in MailLogin). From here, folders can be opened from the account where the Message class objects are stored. These need to be processed first before being able to be displayed in HTML and parsed through to collect its attachments. The messages can be retrieved as long as the Folder class object is opened. That operation will be explained later (along with the "why Outlook is still experimental" explanation). For managing those folders and messages, this is where MailManage comes in.

```
// Fields
private MailTreeItem<String> folderRoot = new MailTreeItem<>( name: "");
private List<Folder> foldersList = new ArrayList<>();
private MailFolderUpdater mailFolderUpdater;
private MailMessage selectedMessage;
private MailTreeItem<String> selectedFolder;
private MailAccount mailAccount;

// Constructor
public MailManage() {
    mailFolderUpdater = new MailFolderUpdater(foldersList);

    // Starts multithreading method from MailFolderUpdater (createTask())
    mailFolderUpdater.start();
}
```

Fig. 17: MailManage fields and constructor

The list of folders for a specific account is stored in the foldersList. MailManage also stores the current selected folder and message in the treeView and tableView. The constructor also starts the MailFolderUpdater service (explained below).

```java
// Methods
public void addMailAccount(MailAccount mail) {
    mailAccount = mail;
    MailTreeItem<String> treeItem = new MailTreeItem<>(mail.getAddress());
    MailFolderFetch mailFolderFetch = new MailFolderFetch(mail.getStore(), treeItem, foldersList);

    // Starts multithreading method from MailFolderFetch (createTask())
    mailFolderFetch.start();

    folderRoot.getChildren().add(treeItem);
}

public void setSelectedRead() {
    // Set selected message in tableView as read
    try {
        selectedMessage.setIsRead(true);
        selectedMessage.getMessage().setFlag(Flags.Flag.SEEN, set: true);
        selectedFolder.decrementUnread();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public void setSelectedUnread() {
    // Set selected message in tableView as unread
    try {
        selectedMessage.setIsRead(false);
        selectedMessage.getMessage().setFlag(Flags.Flag.SEEN, set: false);
        selectedFolder.incrementUnread();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public void deleteSelected() {
    // Delete selected message in tableView, message will now only be visible in All Mail
    try {
        selectedMessage.getMessage().setFlag(Flags.Flag.DELETED, set: true);
        selectedFolder.getMailMessages().remove(selectedMessage);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Fig. 18: MailManage methods

When an email account is added through the addMailAccount method, the treeView will start to get populated by MailTreeItem class objects through the MailFolderFetch service. The MailManage class also handles email / messages that are going to be labelled as read, unread, and is even able to delete messages through the right click context menu.

```java
private void fetchFolders() throws MessagingException {
    // Store folders in a Folder array and iterates through them
    Folder[] folders = store.getDefaultFolder().list();
    handleFolders(folders, folderRoot);
}

private void handleFolders(Folder[] folders, MailTreeItem<String> folderRoot) throws MessagingException {
    // Iterate through folders and add them to folderRoot
    for (Folder folder : folders) {
        foldersList.add(folder);
        MailTreeItem<String> mailTreeItem = new MailTreeItem<>(folder.getName());
        folderRoot.getChildren().add(mailTreeItem);
        folderRoot.setExpanded(true);

        // Get messages from folder
        fetchMessagesFromFolder(folder, mailTreeItem);

        // Continuously look for new messages
        addMessageListener(folder, mailTreeItem);

        // Get subfolders from primary folder
        if (folder.getType() == Folder.HOLDS_FOLDERS) {
            handleFolders(folder.list(), mailTreeItem);
        }
    }
}
```

Fig. 18: MailFolderFetch fetchFolders and handleFolders method

The MailFolderFetch class handles what happens to the folders and the messages inside them. When all the folders are fetched from the passed in Store object, the folders are iterated through to get their messages along with any subfolders that are in each folder. Each folder also gets attached a messageCountListener through the addMessageListener method.

```
private void fetchMessagesFromFolder(Folder folder, MailTreeItem<String> mailTreeItem) {
    Service<Object> fetchMessages = new Service<>() {
        @Override
        protected Task<Object> createTask() {
            return new Task<>() {
                @Override
                protected Object call() throws Exception {
                    // Check if folder has anymore additional subfolders
                    if (folder.getType() != Folder.HOLDS_FOLDERS) {
                        // Get the amount of messages and the messages themselves
                        folder.open(Folder.READ_WRITE);
                        int size = folder.getMessageCount();

                        // Get messages from newest to oldest
                        for (int i = size; i > 0; i--) {
                            mailTreeItem.addMail(folder.getMessage(i));
                        }
                    }
                    return null;
                }
            };
        }
    };
    fetchMessages.start();
}
```

Fig. 19: MailFolderFetch fetchMessagesFromFolder method

Messages from the folder are fetched using this fetchMessagesFromFolder method. In this case, a service is used so that multithreading can happen. Through descending iteration, the folder is run through and each message gets added to the mailTreeItem method addMail.

```java
private void addMessageListener(Folder folder, MailTreeItem<String> mailTreeItem) {
    // Continuously looks for updated messages in specified folder
    folder.addMessageCountListener(new MessageCountListener() {
        @Override
        public void messagesAdded(MessageCountEvent e) {
            // Add new incoming messages to the top of the TableView
            for (int i = 0; i < e.getMessages().length; i++) {
                try {
                    Message current = folder.getMessage( msgnum: folder.getMessageCount() - i);
                    mailTreeItem.addMailTop(current);
                } catch (MessagingException | IOException messagingException) {
                    messagingException.printStackTrace();
                }
            }
        }
        @Override
        public void messagesRemoved(MessageCountEvent e) {
            System.out.println("Message removed: " + e);
        }
    });
}
```

Fig. 20: MailFolderFetch addMessageListener method

This messageCountListener keeps track of how many messages are in a specific folder. It by itself can't update the message count (this is for MailFolderUpdater later) but it can detect when the message count is updated. When there is a new message, it will get added to the top of the tableView.

```java
// Methods
@Override
protected Task<Void> createTask() {
    return new Task<>() {
        @Override
        protected Void call() {
            // Infinite loop
            // Checks the message count of open folders to determine if there are new messages
            while (true) {
                try {
                    Thread.sleep( millis: 5000); // Every 5 seconds
                    for (Folder folder: folders) {
                        if (folder.getType() != Folder.HOLDS_FOLDERS && folder.isOpen()) {
                            folder.getMessageCount();
                        }
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    };
}
```

Fig. 21: MailFolderUpdater createTask method

The MailFolderUpdater only has this one method. As long as the service is started (in the constructor of MailManage), every 5 seconds, all the folders that are open will get their messageCount updated. This will in term trigger the messageCountListener of each folder to add new mail to the top.

3.3.7 MailMessage, MailTreeItem, and MailProcess

So MailTreeItem is the class that is responsible for handling the MailMessage class objects that are going to be displayed on the tableView. But what exactly is the MailMessage class?

```
// Fields

// Use SimpleStringProperty and SimpleObjectProperty for JavaFX TableView
private SimpleStringProperty subject;
private SimpleStringProperty sender;
private SimpleStringProperty recipient;
private SimpleObjectProperty<Date> date;
private List<MimeBodyPart> attachments = new ArrayList<>();
private boolean hasAttachments;
private boolean isRead;
private Message message;
private String content;

// Constructor
public MailMessage(String subject, String sender, String recipient, Date date, boolean isRead, Message message, String content, boole
    this.subject = new SimpleStringProperty(subject);
    this.sender = new SimpleStringProperty(sender);
    this.recipient = new SimpleStringProperty(recipient);
    this.date = new SimpleObjectProperty<>(date);
    this.isRead = isRead;
    this.message = message;
    this.content = content;
    this.hasAttachments = hasAttachments;
}
```

Fig. 22: MailMessage fields and constructor

The MailMessage class holds data for display and properties for determining how it's going to be displayed. It holds SimpleStringProperty for the data that's going to be displayed in the tableView along with a small snippet of the content with the Content string. The hasAttachments and isRead boolean determines the style for display in the tableView as discussed from before.

```
// Methods
public void addAttachment(MimeBodyPart mimeBodyPart) {
    // Only add new attachments
    if (!attachments.contains(mimeBodyPart)) {
        attachments.add(mimeBodyPart);
    }
}
```

Fig. 23: MailMessage addAttachment method

This addAttachment method is also used in MailProcess later on in order to add new attachments to a message. Previous attachments won't be re-added in order to prevent stacking of the same attachments in a single message.

```java
// Methods
private MailMessage fetchMail(Message message) throws MessagingException, IOException {
    // Process Message object to get MailMessage object with all of its properties
    boolean isRead = message.getFlags().contains(Flags.Flag.SEEN);
    MailMessage mailMessage = new MailMessage (
            message.getSubject(),
            message.getFrom()[0].toString(),
            message.getRecipients(MimeMessage.RecipientType.TO)[0].toString(),
            message.getSentDate(),
            isRead,
            message,
            MailProcess.getText(message), // Don't display line breaks,
            MailProcess.hasAttachments(message)
    );

    if (!isRead) {
        incrementUnread();
    }

    return mailMessage;
}

public void addMail(Message message) throws MessagingException, IOException {
    // Adds new mailMessage to the bottom of the table
    MailMessage mailMessage = fetchMail(message);

    mailMessages.add(mailMessage);
}

public void addMailTop(Message message) throws MessagingException, IOException {
    // Adds new mailMessage to the top of the table
    MailMessage mailMessage = fetchMail(message);

    mailMessages.add( index: 0, mailMessage);
}

public void incrementUnread() {
    unreadMessages++;
    updateName();
}

public void decrementUnread() {
    unreadMessages--;
    updateName();
}

private void updateName() {
// Sets the value of the MailTreeItem folder name depending on unreadMessages
    if (unreadMessages > 0) {
        this.setValue((String) (name + " (" + unreadMessages + ")"));
    } else {
        this.setValue(name);
    }
}
```

Fig. 24: MailTreeItem methods

The fetchMail method determines the data to be displayed. Properties like the subject, sender, and recipient of the email is retrieved from the original message, while things like the small content snippet and hasAttachments boolean is handled through the MailProcess class static methods (explained later). Both addMail and addMailTop methods utilize the fetchMail method but only differ in where they append the final message in the mailMessages ObservableList. This is also where the unread messages count gets updated which will in turn, update the folder name on the left to display how many unread messages are there in that specific folder.

Now this is where the message / mail processing happens, the MailProcess class.

```
// Fields
private MailMessage mailMessage;
private WebEngine webEngine;
private StringBuffer stringBuffer;

// Constructor
public MailProcess(WebEngine webEngine) {
    this.webEngine = webEngine;
    this.stringBuffer = new StringBuffer();

    this.setOnSucceeded(event -> display());
}

public void setMailMessage(MailMessage mailMessage) {
    this.mailMessage = mailMessage;
}

// Methods
@Override
protected Task<Void> createTask() {
    return new Task<>() {
        @Override
        protected Void call() {
            try {
                load();
            } catch (Exception e){
                e.printStackTrace();
            }
            return null;
        }
    };
}

private void display() {
    // Display contents of string buffer to webView
    webEngine.loadContent(stringBuffer.toString());
}

private void load() throws MessagingException, IOException {
    // Clear string buffer and set it to a new message
    stringBuffer.setLength(0);
    Message message = mailMessage.getMessage();
    String contentType = message.getContentType();

    if (isSimple(contentType)) {
        stringBuffer.append(message.getContent().toString());
    } else if (isMultipart(contentType)) {
        Multipart multipart = (Multipart) message.getContent();
        loadMulti(multipart, stringBuffer);
    }
}
```

Fig. 25: MailProcess fields, constructor, and some methods

MailProcess also extends the Service class in order to utilize multithreading. When it is started, the load method gets called in which a StringBuffer object gets reset and is modified to include the contents of the message. When load is finished, the setOnSucceeded method gets called to call the display method. The display method just loads the content on the webView on the right of the Main Window.

Processing the message is the difficult part of this process.

```
private boolean isSimple(String contentType) {
    return contentType.contains("TEXT/HTML") || contentType.contains("mixed") || contentType.contains("text");
}

private boolean isMultipart(String contentType) {
    return contentType.contains("multipart");
}

private boolean isPlain(String contentType) {
    return contentType.contains("TEXT/PLAIN");
}

private void loadMulti(Multipart multipart, StringBuffer stringBuffer) throws MessagingException, IOException {
    // Processing for multipart messages
    for (int i = multipart.getCount() - 1; i >= 0; i--) {
        BodyPart bodyPart = multipart.getBodyPart(i);
        String contentType = bodyPart.getContentType();

        if (isSimple(contentType)) {
            stringBuffer.append(bodyPart.getContent().toString());
        } else if (isMultipart(contentType)) {
            Multipart multipart2 = (Multipart) bodyPart.getContent();
            loadMulti(multipart2, stringBuffer);
        } else if (!isPlain(contentType)) {
            MimeBodyPart mimeBodyPart = (MimeBodyPart) bodyPart;
            mailMessage.addAttachment(mimeBodyPart);
        }
    }
}
```

Fig. 26: MailProcess four main methods

The isSimple, isMultipart, and isPlain methods are used to determine what type of message is being processed at a given step. If the message is a simple type to start with, then its contents just gets appended as a string to the StringBuffer object and displayed to the webView. Otherwise, it's a multipart message whose contents need to be processed in order to get the HTML and attachments from the message. This is where the Outlook part of the program is still experimental as most Outlook messages are in multipart / alternative, which the program can't display yet.

```
public static String getText(Part p) throws MessagingException, IOException {
    // Message processing for display on tableView
    if (p.isMimeType("text/*")) {
        String s = (String) p.getContent();

        // Replace line breaks with a single space
        s = s.replaceAll( regex: "\\R",  replacement: " ");

        return s;
    } else if (p.isMimeType("multipart/*")) {
        Multipart mp = (Multipart)p.getContent();
        for (int i = 0; i < mp.getCount(); i++) {
            String s = getText(mp.getBodyPart(i));
            if (s != null) {
                return s;
            }
        }
    }
    return null;
}
// getText method is from https://javaee.github.io/javamail/FAQ, modified to exclude line breaks

public static boolean hasAttachments(Message message) throws MessagingException, IOException {
    // Determines if the message has attachments by seeing the number of BodyPart objects in the message
    if (message.isMimeType("multipart/mixed")) {
        Multipart multipart = (Multipart) message.getContent();
        return multipart.getCount() > 1;
    }
    return false;
}
```

Fig. 27: MailProcess static methods

These 2 public static methods can be called anywhere. The getText method is used to get a simple string from the contents of the message / email without any line breaks to show in the tableView. The hasAttachments method is used to see, without processing, if the message has attachments or not. This is useful in order to

immediately set the style of the message when it is first loaded. Both of these methods are called when MailTreeItem's fetchMail method is called.

3.3.8 MailSend and SendController

The final classes MailSend and SendController handle what happens in the Send Window.

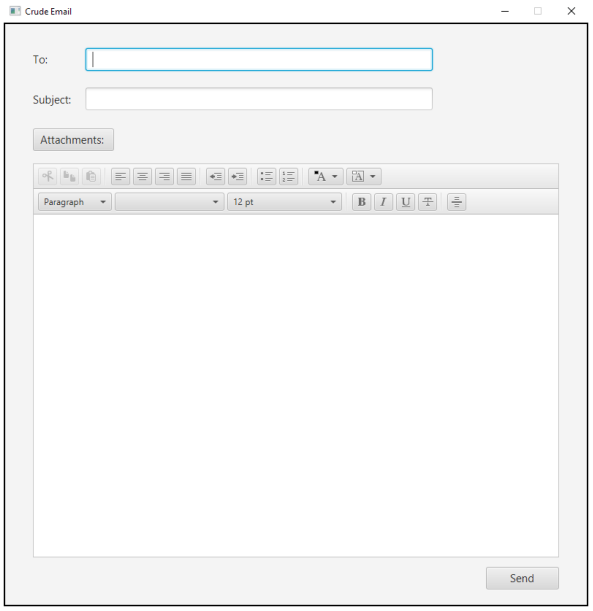


Fig. 28: Empty Send Window

Recipient and subject input fields on top, attachments button, and HTML editor. Once the user has filled in all the data, (attachments are optional), the send button can be clicked to deliver the email.

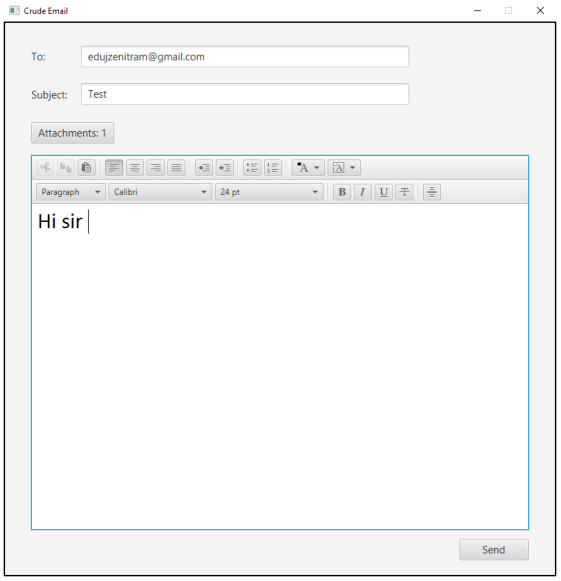


Fig. 29: Filled in Send Window

When attachments are added, the attachments button updates its text according to the number of attachments to be sent. Real time demo of this is in the demo video.

```
private void sendMail() throws MessagingException {
    // Create new message object to send and set some properties
    MimeMessage mimeMessage = new MimeMessage(mailAccount.getSession());
    mimeMessage.setFrom(mailAccount.getAddress());
    mimeMessage.addRecipients(Message.RecipientType.TO, InternetAddress.parse(recipient));
    mimeMessage.setSubject(subject);

    // Set content to be sent
    Multipart multipart = new MimeMultipart();
    BodyPart bodyPart = new MimeBodyPart();
    bodyPart.setContent(content, type: "text/html; charset=utf-8");
    multipart.addBodyPart(bodyPart);
    mimeMessage.setContent(multipart);

    // Attachment handling
    if (!attachments.isEmpty()) {
        for (File file: attachments) {
            // Create new MimeBodyPart to be added to multipart for each attachment
            MimeBodyPart mimeBodyPart = new MimeBodyPart();
            FileDataSource source = new FileDataSource(file.getAbsolutePath());
            mimeBodyPart.setDataHandler(new DataHandler(source));
            mimeBodyPart.setFileName(file.getName());
            multipart.addBodyPart(mimeBodyPart);
        }
    }

    // Send the final message
    Transport transport = mailAccount.getSession().getTransport();
    transport.connect(
            mailAccount.getProp().getProperty("mail.smtp.host"),
            mailAccount.getAddress(),
            mailAccount.getPassword()
    );

    transport.sendMessage(mimeMessage, mimeMessage.getAllRecipients());
    transport.close();
}
```

Fig. 30: MailSend sendMail method

The sendMail method in MailSend is called upon in the SendController. This method is wrapped in the createTask method as a service that is started. Data is extracted from the passed in arguments during its instantiation. Once it finishes, the Transport class object sends the message off by first connecting to the SMTP sending server of the email account and finally sending the message before being closed again.

```java
@FXML
void sendMail() {
    MailSend mailSend = new MailSend(
            mailManage.getMailAccount(),
            recipientInput.getText(),
            subjectInput.getText(),
            contentInput.getHtmlText(),
            attachments
    );

    // Starts multithreading method from MailSend (createTask())
    mailSend.start();
    // Disable send button and make progress bar visible
    sendProgressBar.setOpacity(1);
    sendMailButton.setDisable(true);

    mailSend.setOnSucceeded(event -> {

        String result = mailSend.getValue();

        switch (result) {
            case "SUCCESS":
                Stage currentStage = (Stage) recipientInput.getScene().getWindow();
                resourcesController.closeStage(currentStage);
                break;
            case "UNEXPECTED ERROR":
                sendErrorLabel.setText("Unexpected Error");
                break;
            case "NETWORK FAILED":
                sendErrorLabel.setText("Network Error");
                break;
        }
        // Re-enable send button and make progress bar invisible
        sendMailButton.setDisable(false);
        sendProgressBar.setOpacity(0);
    });
}
```

Fig. 31: SendController sendMail method

A separate sendMail method is called when the send button is clicked in the Send Window. This is where the MailSend object gets its data as passed in arguments. Data is extracted from the input fields, HTML editor, and the attachments button. If the value returned is successful, then the window will close, indicating that the message is sent. Otherwise, error labels similar to the ones in the Login Window are brought up.

```java
@FXML
void attachmentsButtonAction(ActionEvent event) {
    // Determine what happens when attachment button is pressed
    FileChooser fileChooser = new FileChooser();
    File file = fileChooser.showOpenDialog( ownerWindow: null);

    // Adds file from FileChooser and increment the attachment button label
    if (file != null) {
        attachments.add(file);
        attachmentsButton.setText("Attachments: " + attachments.size());
    }
}
```

Fig. 32: SendController attachmentsButtonAction method

The attachments button utilize the FileChoose class object in order for the user to pick and choose which attachments that they want to add to the email.

4. Resources and References

- JavaFX and SceneBuilder
  - https://openjfx.io/
  - https://gluonhq.com/products/scene-builder/
  - http://tutorials.jenkov.com/javafx/webview.html
  - https://www.youtube.com/playlist?list=PLrzWQu7Ajpi26jZvP8JhEJgFPFEj_fojO
  - https://www.reddit.com/r/javahelp/comments/84w6i6/problem_displaying_anything_with_javafx_only/
- Jakarta Mail
  - https://eclipse-ee4j.github.io/mail/docs/api/
  - https://www.javatpoint.com/java-mail-api-tutorial
  - https://eclipse-ee4j.github.io/mail/FAQ
- Email in general
  - https://www.makeuseof.com/tag/pop-vs-imap/
  - https://kinsta.com/knowledgebase/tls-vs-ssl/
  - https://support.google.com/mail/answer/7126229?hl=en
  - https://support.microsoft.com/en-us/office/pop-imap-and-smtp-settings-8361e398-8af4-4e97-b147-6c6c4ac95353
- Maven
  - https://www.youtube.com/watch?v=d4FMEgjSdEw