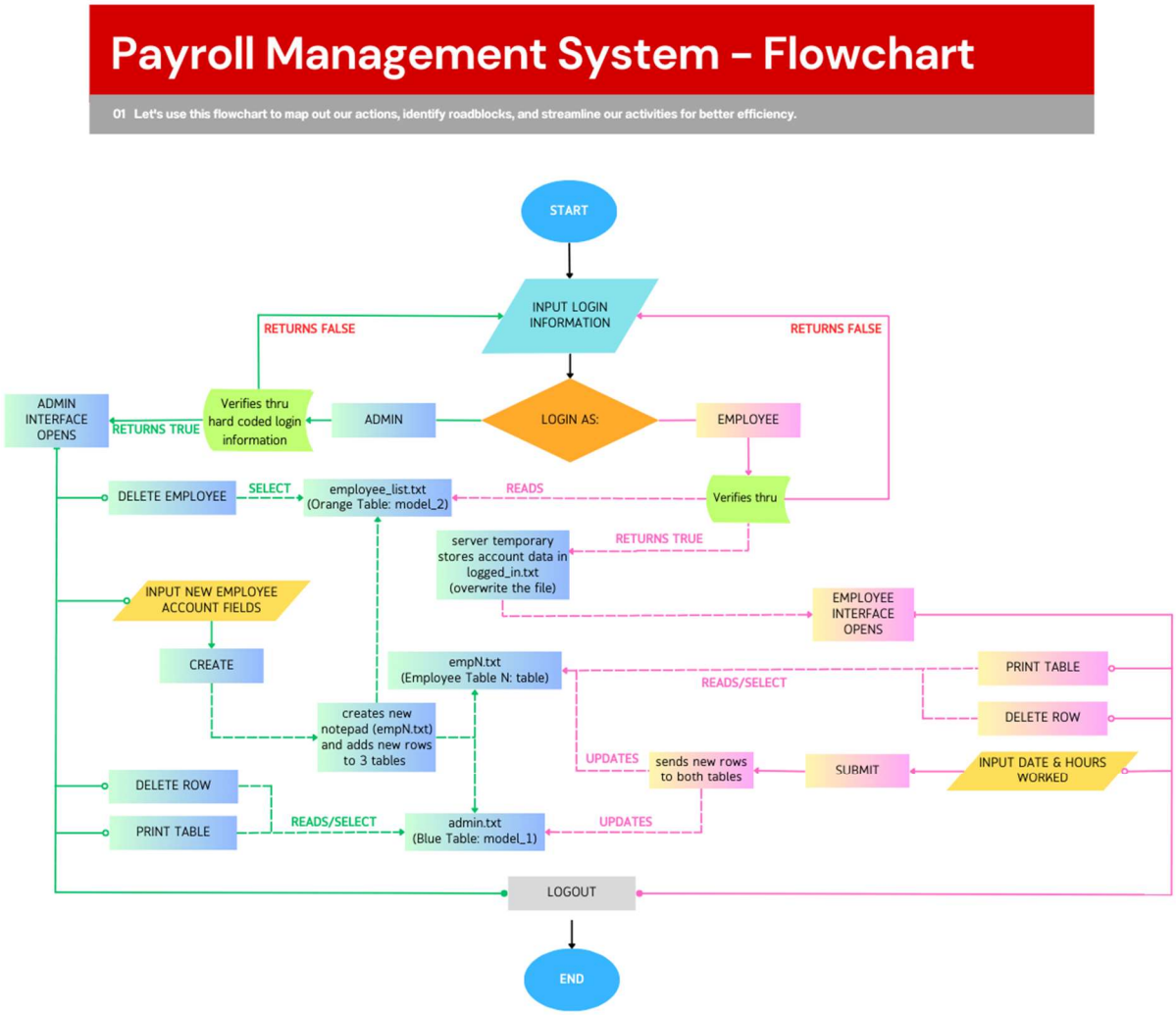
PSM DOCUMENTATION

Introduction:

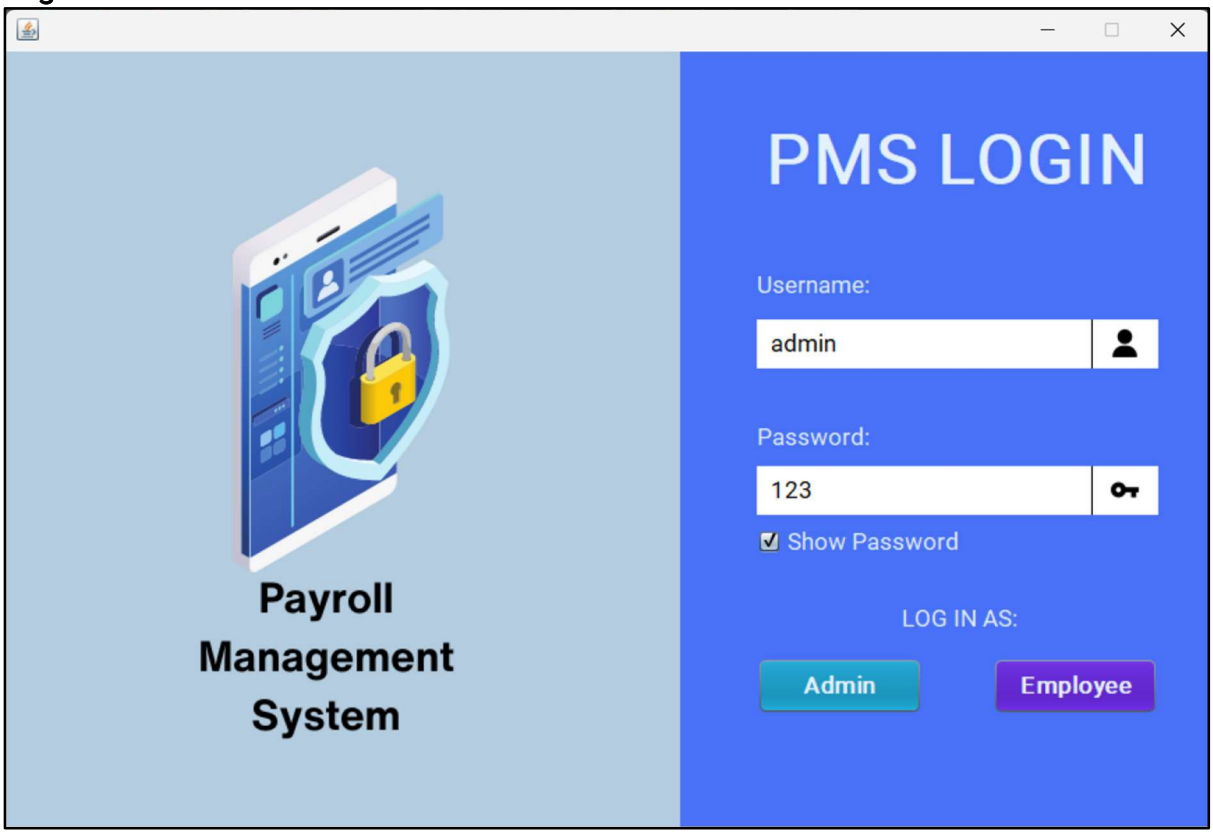
Welcome to my documentary report on the Payroll Management System, a groundbreaking project designed to streamline operations for managers and employees alike. This system has been meticulously crafted to ensure a user-friendly experience, catering to both managers and employees within their respective workplace environments. Our code not only empowers managers to effortlessly modify and access comprehensive work history records of employees but also enables employees to efficiently log their work hours and calculate their earned rates. Join us as we delve into the intricacies of this innovative system, revolutionizing payroll management in the modern era of programming.

Flowchart

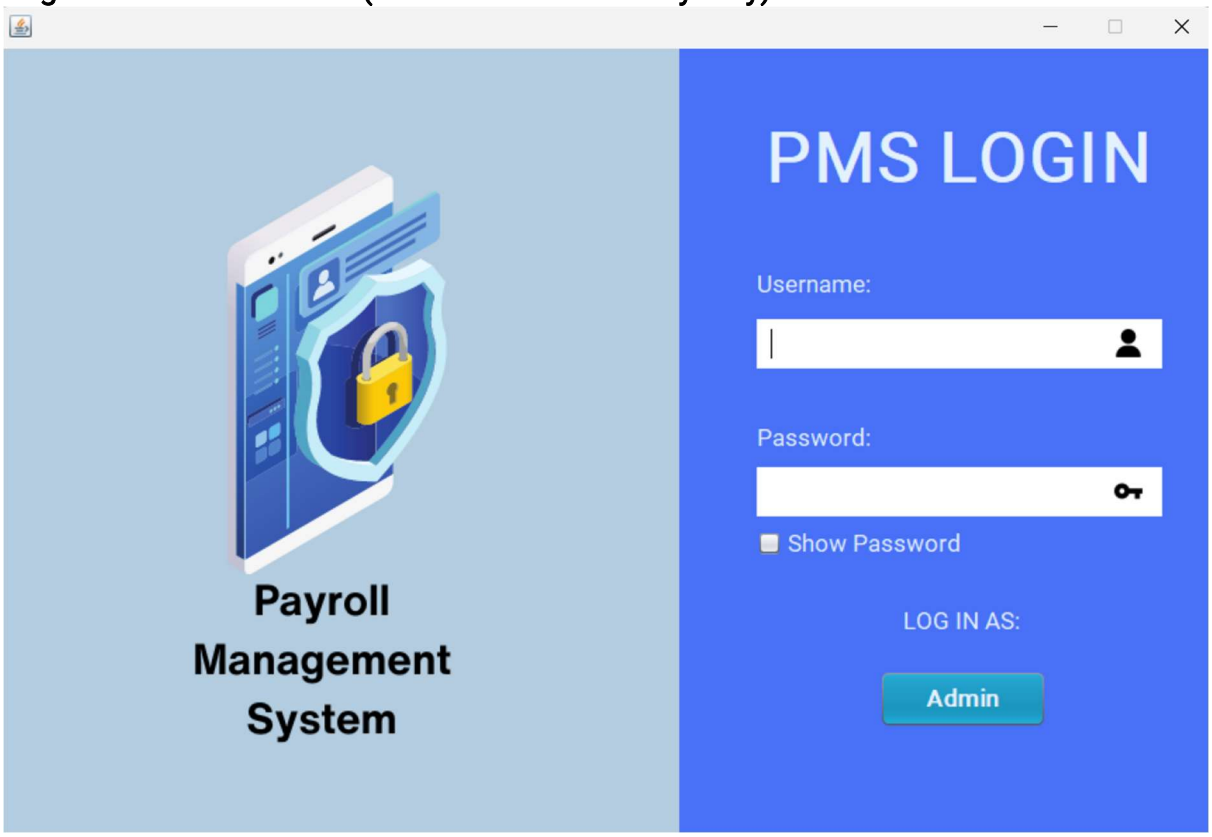


Flowchart link:
https://www.canva.com/design/DAFjOj02cuM/cTBJBFdBQtE5i4gULBCjiA/edit?utm_content=DAFjOj02cuM&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

Login Interface UI Client:



Login Interface UI Server (for admin accessibility only):



LOGIN INTERFACE CLIENT CODES:

Code Snippet:

```
        private static final String SERVER_IP = "192.168.1.1";
        private static final int SERVER_PORT = 5000;
```

SERVER_IP
Type: String
Purpose: Store IP address of the server
Description: We initialize it at this code at the top of the code and we will use this SERVER_IP to communicate from client to server.

SERVER_PORT
Type: Integer
Purpose: Represents the port number of the server.
Description: It holds the port number 5000 which is a recommended and safe range in order to communicate with the server.

Code Snippet

```
try {
  for (LookAndFeelInfo info : UIManager.getInstalledLookAndFeels())
{
    if ("Nimbus".equals(info.getName())) {
      UIManager.setLookAndFeel(info.getClassName());
      break;
    }
  }
```

Type: try-catch statement
Purpose: Display 3d interface
Description: This design embodies elegance looks and make buttons 3d.

Code Snippet:

```
    chckbxShowPassword.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        char echoChar = chckbxShowPassword.isSelected() ? '\0' :
'•';
        pwdPassword.setEchoChar(echoChar);
      }
    });
```

Type: Event Listener
Purpose: Allows toggling password visibility
Description: When the checkbox is clicked, it triggers the actionPerformed method. Within this method, a character variable (echoChar) is assigned either a null character ('\0') or a bullet character ('•') based on the checkbox's selected state. The echoChar value is then used to control the visibility of the password field (pwdPassword) by setting its echo character. If the checkbox is selected, the echo character is null, revealing the password text. Otherwise, the echo character is set to a bullet, hiding the password. This functionality enhances user experience by enabling the display or concealment of sensitive password information based on the checkbox selection.

Code Snippet:

```
btnAdmin.addActionListener(new ActionListener() {
  @SuppressWarnings("deprecation")
  public void actionPerformed(ActionEvent e) {
    if (txtUsername.getText().equals("admin") &&
pwdPassword.getText().equals("123")) {
      JOptionPane.showMessageDialog(null, "Login successful!");
      AdminInterface admin = new AdminInterface();
      admin.setVisible(true);
      dispose();
    } else {
      JOptionPane.showMessageDialog(null, "Invalid username or password!", "Invalid
credentials", JOptionPane.ERROR_MESSAGE);
      pwdPassword.setText("");
    }
  }
});
```

Type: Event Listener

Purpose: Checks whether hard coded admin and 123 is true
Description: When the button is clicked, it triggers the actionPerformed method. Within this method, the code checks if the entered username (txtUsername.getText()) is equal to "admin" and the entered password (pwdPassword.getText()) is equal to "123". If the condition is true, a success message is displayed using a message dialog and AdminInterface object is created, set as visible, and the current interface is disposed. On the other hand, if the condition is false, an error message is displayed with a message dialog stating invalid credentials. The password field (pwdPassword) is cleared for another login attempt.

Code Snippet:

```
    btnEmployee.addActionListener(new ActionListener() {
      @SuppressWarnings("deprecation")
      public void actionPerformed(ActionEvent e) {
        if (validateLogin(txtUsername.getText(), pwdPassword.getText())) {
          JOptionPane.showMessageDialog(null, "Login successful!");
          EmployeeInterface employee = new EmployeeInterface();
          employee.setVisible(true);
          dispose();
        } else {
          JOptionPane.showMessageDialog(null, "Invalid username or password!",
"Invalid credentials",
              JOptionPane.ERROR_MESSAGE);
          pwdPassword.setText("");
        }
      }
    });
```

Type: Event Listener
Purpose: Checks whether validateLogin parameter is true
Description: When the button is clicked, it triggers the "actionPerformed" method. Inside this method, the code validates the login credentials entered in the "txtUsername" and "pwdPassword" fields using "validateLogin" method. If the credentials are valid, a success message is displayed through a message dialog. An "EmployeeInterface" object is created, set as visible, and the current interface is disposed. If the credentials are invalid, an error message is displayed using a message dialog, indicating invalid credentials. The password field is cleared for another login attempt.

Code Snippet:

```
    private boolean validateLogin(String username, String password) {
        try (Socket socket = new Socket(SERVER_IP, SERVER_PORT)) {
            PrintWriter writer = new PrintWriter(new
OutputStreamWriter(socket.getOutputStream()), true);
            BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

            writer.println("LOGIN");
            writer.println(username);
            writer.println(password);

            String response = reader.readLine();

            writer.close();
            reader.close();

            return "SUCCESS".equals(response);
        } catch (IOException e) {
            e.printStackTrace();
            return false;
        }
    }
```

Type: Method
Purpose: Validates login credentials by communicating with the server.
Description: A private method "validateLogin" with username and password as input parameters. It establishes a socket connection with the server using the "SERVER_IP" and "SERVER_PORT”. The method creates a PrintWriter and BufferedReader to send and receive data from the server. It sends the login request by writing "LOGIN" to the server, followed by the username and password. The server's response is read from the BufferedReader and stored in the "response" variable. The PrintWriter and BufferedReader are then closed. If the response is equal to "SUCCESS", the method returns true, indicating a successful login. If an IOException occurs during the communication with the server, the method prints the stack trace, indicating a failure in the login validation process, and returns false.

LOGIN INTERFACE SERVER CODES:

Code Snippet:

```
public class Server {
  private static final int PORT = 5000; // Choose a suitable port number

  public static void main(String[] args) {
    try (ServerSocket serverSocket = new ServerSocket(PORT)) {
      System.out.println("Server started. Waiting for client connection...");

      while (true) {
        Socket clientSocket = serverSocket.accept();
        System.out.println("Client connected: " +
clientSocket.getInetAddress().getHostAddress());
        PrintWriter writerLogin = new PrintWriter(clientSocket.getOutputStream(),
true);

        BufferedReader reader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

        String requestType = reader.readLine();
```

Type: Code Block
Purpose: Facilitates server-client communication.
Description: Inside the try block ServerSocket is established which listens on a inputted port number. The server continuously waits for client connections by accepting clientSocket. When a client connects, the server prints the client's IP address. Then PrintWriter is used to send data to the client and a BufferedReader to receive data from the client's input stream. This also reads the request type of the client string requestType.

Code Snippet:

```
        PrintWriter writerLogin = new PrintWriter(clientSocket.getOutputStream(),
true);
        BufferedReader reader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

        String requestType = reader.readLine();

        if ("LOGIN".equals(requestType)) {
          String username = reader.readLine();
          String password = reader.readLine();
          if (validateLogin(username, password)) {
            writerLogin.println("SUCCESS");
          } else {
            writerLogin.println("FAILURE");
          }
        } else if
```

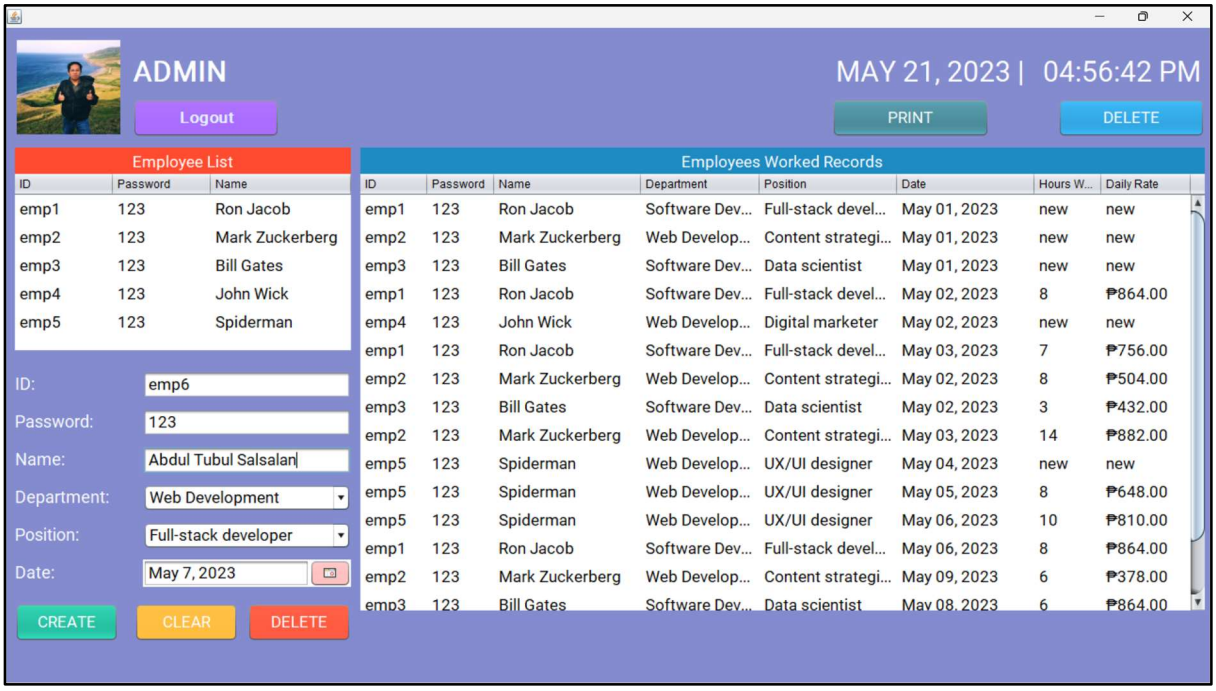
Type: Code Block
Purpose: Handles client requests for login verification.
Description: The PrintWriter class is utilize to send data to the client and a BufferedReader to receive data from the client's input stream. The code reads the request type sent by the client. If the request type is "LOGIN", the code proceeds to read the username and password from the client. It then validates the login credentials using a separate "validateLogin" method. If the credentials are valid, the code sends the message "SUCCESS" back to the client using the PrintWriter. Otherwise, it sends "FAILURE"

Code Snippet:

```
  private static boolean validateLogin(String username, String password) {
    try {
      BufferedReader reader = new BufferedReader(new
FileReader("datas/employee_list.txt"));

      String line;
      while ((line = reader.readLine()) != null) {
        String[] index = line.split("// ");

        String storedID = index[0];
        String storedPassword = index[1];
        if (username.equals(storedID) && password.equals(storedPassword)) {
          FileWriter writer = new FileWriter("datas/logged_in.txt");
          writer.write("");
          writer.write(line);
          writer.close();

          reader.close();
          return true;
        }
      }
      reader.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
    return false;
  }
```

Type: Method
Purpose: Validates login credentials by reading from a txt file.
Description: s defined static method named "validateLogin" that takes a username and password as input parameters. It attempts to read from a file named "employee_list.txt" located in the "datas" directory. Inside a try-catch statekejt, a BufferedReader is initialize to read the datas of the file. The code reads each line from the file and splits it into an array using the "// " split method. The username and password stored in each line are compared with the provided username and password. If a match is found, the FileWriter class will write the matched line to a file named "logged_in.txt" to store who's logged in. The FileWriter is then closed, and the method returns true. If no match is found or an IOException occurs during file reading, the method prints the stack trace and returns false.

Admin Interface UI:



ADMIN INTERFACE CLIENT CODES:

Code Snippet:

```
    btnLogOut.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        int choice = JOptionPane.showConfirmDialog(null, "Do you want to proceed?",
"Confirmation",
            JOptionPane.YES_NO_OPTION);
        if (choice == JOptionPane.YES_OPTION) {
          System.exit(0);
        }
      }
```

Type: Event Listener
Purpose: Logout the interface.
Description: When the "btnLogOut" is clicked. An ActionLisstener is assigned to the button. When the button is clicked, a confirmation dialog is displayed using JOptionPane. The user is asked if they want to proceed with the logout. If the user selects "Yes" (JOptionPane.YES_OPTION), the program exits by calling "System.exit(0);"

Code Snippet:

```
    cbDepartment.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        cbPosition.setEnabled(true);
        Object selectedDepartment = cbDepartment.getSelectedItem();
        if (selectedDepartment != null && selectedDepartment.equals("Software
Development")) {
          cbPosition.setModel(new DefaultComboBoxModel(new String[] { "Front-end
developer", "Back-end developer", "Full-stack developer", "QA engineer", "DevOps
engineer", "Technical writer", "Database administrator", "Data scientist",
"Security specialist" }));
        } else {
          cbPosition.setModel(new DefaultComboBoxModel(new String[] { "Front-end
developer", "Back-end developer", "Full-stack developer", "UX/UI designer", "QA
engineer", "DevOps engineer", "Technical writer", "SEO specialist", "Content
strategist", "Digital marketer", "Analytics specialist" }));
        }
      }
    });
```

Type: Event Listener
Purpose: Updates positions based on selected department combobox
Description: When cbDepartment is selected, the code enables the "cbPosition" ComboBox. It retrieves the selected department using "cbDepartment.getSelectedItem()".

If the selected department is "Software Development", the "cbPosition" ComboBox is populated with options specific to software development roles, such as front-end developer, back-end developer, full-stack developer, etc. Otherwise, if the selected department is “Web Development”, the "cbPosition" ComboBox is populated with a options like UX/UI designer, SEO specialist, digital marketer, etc.

Code Snippet:

```
    btnClear.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        int choice = JOptionPane.showConfirmDialog(null, "Do you want to proceed?",
"Confirmation",
            JOptionPane.YES_NO_OPTION);
        if (choice == JOptionPane.YES_OPTION) {
          txtID.setText("");
          txtPassword.setText("");
          txtName.setText("");
          cbDepartment.setSelectedIndex(-1);
          cbPosition.setSelectedIndex(-1);
          cbPosition.setEnabled(false);
        }
      }
    });
```

Type: Code Block
Purpose: Clears txtfields
Description: When the button is clicked, the code displays a confirmation dialog box using JOptionPane. If the user chooses to proceed (by selecting "Yes" in the dialog), it then set the text of txtfields to empty string and combobox index to -1 and specified cbPosition to enabled false prompting new inputs.

Code Snippet:

```
  public String date() {
    Date date = new Date();
    SimpleDateFormat sdf = new SimpleDateFormat("MMM dd,
yyyy");
    String dateOutput = sdf.format(date).toUpperCase();

    return dateOutput;
  }
```

Type: Method
Purpose: Retrieve the current date with specific formatting.
Description: This date() method returns the current date in a specific format.

Inside the method, it creates a new Date class to represent the current date and time. Then implemented a SimpleDateFormat object, "sdf," with the desired date format pattern "MMM dd, yyyy" (eg., MAY 23, 2023).

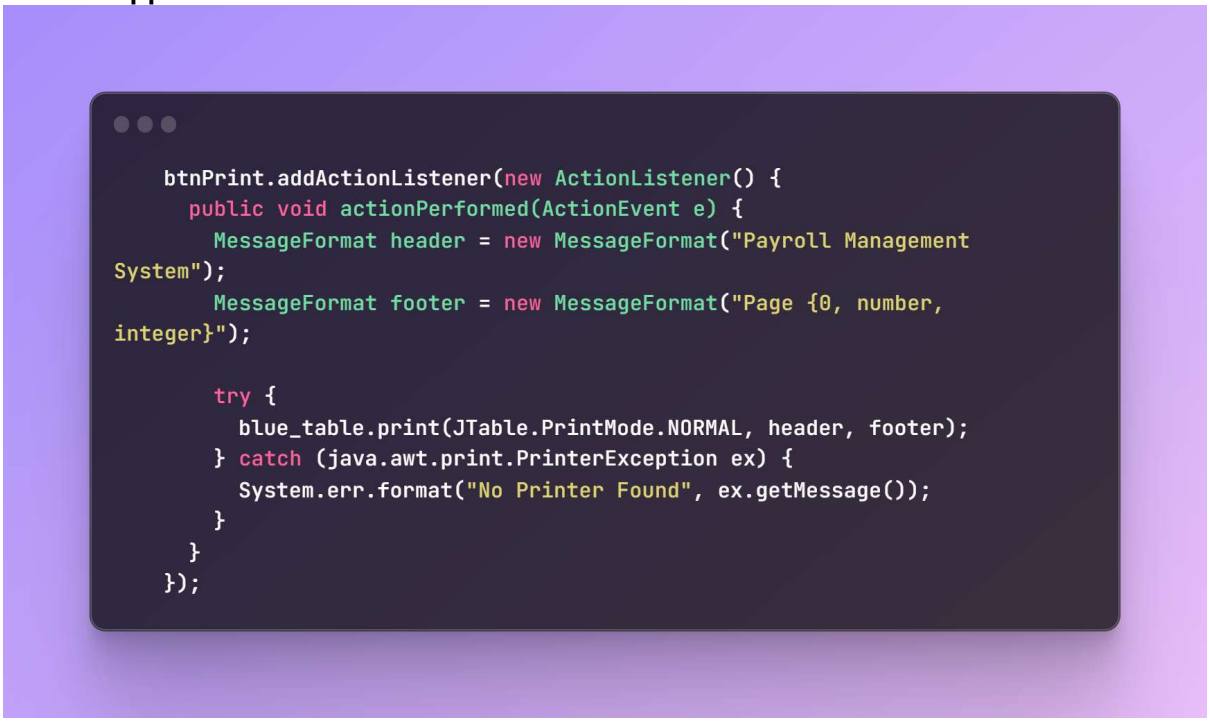
The method formats the date object using the SimpleDateFormat object and assigns the formatted date string to the variable "dateOutput". It converts the formatted date to uppercase letters and returns it.

Code Snippet:

```
  public void time(JLabel lblTime) {
    timeFormat = new SimpleDateFormat("hh:mm:ss
a");
    timer = new Timer(1000, new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
        Date now = new Date();
        String timeStr = timeFormat.format(now);
        lblTime.setText(timeStr);
      }
    });
    timer.setInitialDelay(0);
    timer.start();
  }
```

Type: Real-time Clock
Purpose: Update a label with the current time in a specific format.
Description: This method sets up a real-time clock by creating a timer that updates the label (setText) with the current time every second. It uses the specified time format ("hh:mm:ss a") to format the time string. The timer is started with an initial delay of 0 milliseconds.

Code Snippet:



```
    btnPrint.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        MessageFormat header = new MessageFormat("Payroll Management
System");
        MessageFormat footer = new MessageFormat("Page {0, number,
integer}");

        try {
          blue_table.print(JTable.PrintMode.NORMAL, header, footer);
        } catch (java.awt.print.PrinterException ex) {
          System.err.format("No Printer Found", ex.getMessage());
        }
      }
    });
```

Type: Code Block
Purpose: Print the blue table.
Description: When the button is clicked, the code creates a header and footer message format for the printed document. It attempts to print the "blue_table" JTable using the "print" method, with the specified print mode, header, and footer. If a printer is found, the table is printed with the provided header and footer. If no printer is found, an error message is displayed.

Code Snippet:

```
    btnCreate.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        Date selectedDate = dateChooser.getDate();

        if (txtID.getText().equals("") || txtPassword.getText().equals("") ||
txtName.getText().equals("")
            || cbDepartment.getSelectedItem() == null ||
cbPosition.getSelectedItem() == null
            || selectedDate == null) {
          JOptionPane.showMessageDialog(null, "Please fill in the blanks");
        } else {
          try (Socket socket = new Socket(SERVER_IP, SERVER_PORT)) {
            SimpleDateFormat dateFormat = new SimpleDateFormat("MMM dd, yyyy");
            String formattedDate = dateFormat.format(selectedDate);

            PrintWriter writer = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

            writer.println("CREATE");
            writer.println(txtID.getText());
            writer.println(txtPassword.getText());
            writer.println(txtName.getText());
            writer.println(cbDepartment.getSelectedItem().toString());
            writer.println(cbPosition.getSelectedItem().toString());
            writer.println(formattedDate);

            String response = reader.readLine();

            if ("SUCCESS".equals(response)) {
              Object[] row = new Object[8];
              row[0] = txtID.getText();
              row[1] = txtPassword.getText();
              row[2] = txtName.getText();
              row[3] = cbDepartment.getSelectedItem();
              row[4] = cbPosition.getSelectedItem();
              row[5] = formattedDate;
              row[6] = "new";
              row[7] = "new";
              model_1.addRow(row);

              Object[] row_1 = new Object[3];
              row_1[0] = txtID.getText();
              row_1[1] = txtPassword.getText();
              row_1[2] = txtName.getText();
              model_2.addRow(row);

              JOptionPane.showMessageDialog(null, "Record created successfully.");
            } else if ("ID_EXISTS".equals(response)) {
              JOptionPane.showMessageDialog(null, "ID already exists.");
            } else {
              JOptionPane.showMessageDialog(null, "Failed to create record.");
            }
            reader.close();
            writer.close();
          } catch (IOException e1) {
            e1.printStackTrace();
          }
          txtID.setText("");
          txtPassword.setText("");
          txtName.setText("");
          cbDepartment.setSelectedIndex(-1);
          cbPosition.setSelectedIndex(-1);
          cbPosition.setEnabled(false);
        }
      }
    });
```

Type: Code Block

Purpose: Create new employee and a record to blue and orange table
Description: When the button is clicked, the code retrieves the selected date from the "dateChooser" and validates if required fields are filled.

If any of the fields are missing, a message dialog is displayed prompting the user to fill in the blanks. Otherwise, the code establishes a socket connection with the specified server IP and port. It sends a "CREATE" request along with the entered employee details to the server and waits for a response.

If the response is "SUCCESS", indicating that the record creation was successful, the code adds the new employee's information to both "model_1" and "model_2" and it displays a success message, and clears the input fields. If the response is "ID_EXISTS", it displays an error message indicating that the ID already exists. For any other response, a failure message is shown.

Code Snippet:

```
    btnDeleteBlueTableRow.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        int selectedRow = blue_table.getSelectedRow();

        if (selectedRow >= 0) {
          int choice = JOptionPane.showConfirmDialog(null, "Do you want to
proceed?", "Confirmation",
              JOptionPane.YES_NO_OPTION);
          if (choice == JOptionPane.YES_OPTION) {
            try (Socket socket = new Socket(SERVER_IP, SERVER_PORT)) {
              PrintWriter writer = new PrintWriter(socket.getOutputStream(), true);

              writer.println("DELETE_BLUE_TABLE_ROW");
              writer.println(selectedRow);

              model_1.removeRow(selectedRow);
              JOptionPane.showMessageDialog(null, "Deleted Successfully");
            } catch (IOException e1) {
              e1.printStackTrace();
            }
          }
        } else {
          JOptionPane.showMessageDialog(null, "Select a row first");
        }
      }
    });
```

```
    btnDeleteOrangeTableRow.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        int selectedRowOrangeTable = orange_table.getSelectedRow();

        if (selectedRowOrangeTable >= 0) {
          int choice = JOptionPane.showConfirmDialog(null, "Do you want to
proceed?", "Confirmation",
              JOptionPane.YES_NO_OPTION);
          // YES == 0
          // NO == 1
          if (choice == JOptionPane.YES_OPTION) {
            try (Socket socket = new Socket(SERVER_IP, SERVER_PORT)) {
              PrintWriter writer = new PrintWriter(socket.getOutputStream(), true);

              writer.println("DELETE_ORANGE_TABLE_ROW");
              writer.println(selectedRowOrangeTable);

              model_2.removeRow(selectedRowOrangeTable);
              JOptionPane.showMessageDialog(null, "Deleted Successfully");
            } catch (IOException e1) {
              e1.printStackTrace();
            }
          }
        } else {
          JOptionPane.showMessageDialog(null, "Select a row first");
        }
      }
    });
```

Type: Event Listener
Purpose: Delete a selected row from the table.
Description: When clicked, it checks if a row is selected and prompts the user for confirmation. If confirmed, it establishes a connection to the server and sends a request to delete the corresponding row in the table. The selected row is then removed from the table model, and a success message is displayed upon successful deletion. If no row is selected, a “Select a row first” is shown.

Code Snippet:

```
      loadBlueTable(model_1);
  }
    public void loadBlueTable(DefaultTableModel model) {
    try (Socket socket = new Socket(SERVER_IP, SERVER_PORT)) {
      PrintWriter writer = new PrintWriter(new
OutputStreamWriter(socket.getOutputStream()), true);

      writer.println("BLUE_TABLE");
      BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
      String line;
      while ((line = reader.readLine()) != null) {
        String[] fields = line.split("// ");
        Object[] row = new Object[8];
        row[0] = fields[0]; // ID
        row[1] = fields[1]; // PASSWORD
        row[2] = fields[2]; // NAME
        row[3] = fields[3]; // DEPARTMENT
        row[4] = fields[4]; // POSITION
        row[5] = fields[5]; // DATE
        row[6] = fields[6]; // HOURS WORKED

        String regex = "\\d+";
        String text = fields[7];

        Pattern pt = Pattern.compile(regex);
        Matcher mt = pt.matcher(text);
        boolean result = mt.find();

        if (result) {
          row[7] = "\u20B1" + fields[7]; // DAILY RATE
        } else {
          row[7] = fields[7]; // new
        }
        model.addRow(row);
      }
    } catch (IOException e1) {
      e1.printStackTrace();
    }
  }
```

Type: Method
Purpose: Load data into the blue_table using the model.
Description: First, it establishes a connection to the server using the SERVER_IP and SERVER_PORT. Then, it creates a PrintWriter object, "writer," to send data to the server through the socket's output stream. It sends the string "BLUE_TABLE" to indicate the request type.

Next, it creates a BufferedReader object, "reader," to read data from the server through the socket's input stream.

It enters a loop where it reads each line of data from the server using reader.readLine(). The data is split into fields using the "// " delimiter.
For each line, it creates an Object array, "row," with a length of 8 to store the fields' values such as ID, PASSWORD, NAME, DEPARTMENT, POSITION, DATE, HOURS WORKED, DAILY RATE. The values from the fields array are assigned to the corresponding column names in the row array. It performs a regex check on fields[7] to determine if it contains only digits. If it does, it formats it as a currency value by appending the peso symbol ("\u20B1") before the value. Otherwise, it leaves it as is. Finally, the row array is added to the model using model.addRow(row).

This method is used to populate the blue_table with data received from the server.

Code Snippet:

```
    loadOrangeTable(model_2);
}
      public void loadOrangeTable(DefaultTableModel model) {
    try (Socket socket = new Socket(SERVER_IP, SERVER_PORT)) {
      PrintWriter writer = new PrintWriter(new
OutputStreamWriter(socket.getOutputStream()), true);

      writer.println("ORANGE_TABLE");
      BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
      String line;
      while ((line = reader.readLine()) != null) {
        String[] fields = line.split("// ");
        Object[] row = new Object[3];
        row[0] = fields[0]; // ID
        row[1] = fields[1]; // PASSWORD
        row[2] = fields[2]; // NAME
        model.addRow(row);
      }
    } catch (IOException e1) {
      e1.printStackTrace();
    }
  }
```

Type: Method
Purpose: Load data into the orange_table using the model.
Description: First, it establishes a connection to the server using the SERVER_IP and SERVER_PORT. Then, it creates a PrintWriter object, "writer," to send data to the server through the socket's output stream. It sends the string "ORANGE_TABLE" to indicate the request type.

Next, it creates a BufferedReader object, "reader," to read data from the server through the socket's input stream. it enters a loop where it reads each line of data from the server using reader.readLine(). The data is split into fields using the "// " delimiter. For each line, it creates an Object array, "row," with a length of 3 to store

the fields' values. Finally, the row array is added to the model using model.addRow(row).

This method is used to populate the orange_table with data received from the server.

ADMIN INTERFACE SERVER CODES:

```
} else if ("BLUE_TABLE".equals(requestType)) {
    sendBlueTableData("datas/admin.txt", writerLogin);
} else if ("ORANGE_TABLE".equals(requestType)) {
    sendOrangeTableData("datas/employee_list.txt", writerLogin);
} ...

private static void sendBlueTableData(String filePath, PrintWriter writer) {
    try (BufferedReader reader = new BufferedReader(new FileReader(filePath)))
{
      String line;
      while ((line = reader.readLine()) != null) {
        writer.println(line);
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

private static void sendOrangeTableData(String filePath, PrintWriter writer) {
    try (BufferedReader reader = new BufferedReader(new FileReader(filePath)))
{
      String line;
      while ((line = reader.readLine()) != null) {
        writer.println(line);
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
```

Type: Method
Purpose: Sending blue/orange table data to the client.
Description: When request type is received, this method with parameter Path and PrintWriter will be execued. This method reads the contents of the "admin.txt" or “employee_list.txt” file and sends each line to the client through a provided PrintWriter object. It handles any potential IOException that may occur during the file reading process.

Code Snippet:

```
        } else if ("CREATE".equals(requestType)) {
          String id = reader.readLine();
          String password = reader.readLine();
          String name = reader.readLine();
          String department = reader.readLine();
          String position = reader.readLine();
          String formattedDate = reader.readLine();

          PrintWriter writerCreate = new
PrintWriter(clientSocket.getOutputStream(), true);

          File notepadFile = new File("employees/" + id + ".txt");
          if (notepadFile.exists()) {
            writerCreate.println("ID_EXISTS");
          } else {
            notepadFile.createNewFile();
            writerCreate.println("SUCCESS");

            FileWriter writer1 = new FileWriter("employees/" + id + ".txt");
            writer1.write(id + "// " + password + "// " + name + "// " + department
+ "// " + position  + "// " + formattedDate + "// new// new\n");
            writer1.close();

            FileWriter writer2 = new FileWriter("datas/employee_list.txt", true);
            writer2.write(id + "// " + password + "// " + name + "// " + department
+ "// " + position + "// " + formattedDate + "// new// new\n");
            writer2.close();

            FileWriter writer3 = new FileWriter("datas/admin.txt", true);
            writer3.write(id + "// " + password + "// " + name + "// " + department
+ "// " + position + "// " + formattedDate + "// new// new\n");
            writer3.close();
          }
        }
        clientSocket.close();
        System.out.println("Client disconnected.");
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
```

Type: Code Block
Purpose: Handles the "CREATE" request from the client.
Description: When the server receives a "CREATE" request, it reads the employee details (ID, password, name, department, position, and formatted date) from the input stream using the "reader" object.
The code then creates a PrintWriter object, "writerCreate," to send the response back to the client. If a file with the same ID already exists in the "employees/ID.txt file path, the server sends the response "ID_EXISTS" to indicate that the ID is already taken. Otherwise, it proceeds to create a new file with the employee's ID as its filename.

If the file creation is successful, the server sends the response "SUCCESS" to indicate that the record creation was successful. It then writes the employee's details to the newly created file and appends the same details to both "employee_list.txt" and "admin.txt" files.

After the request, the server closes the client socket and prints a message indicating that the client has disconnected.

Code Snippet:

```
} else if ("DELETE_BLUE_TABLE_ROW".equals(requestType)) {
          int selectedRowBlueTable =
Integer.parseInt(reader.readLine());
          try {
            Path path = Paths.get("datas/admin.txt");
            List<String> lines = Files.readAllLines(path);

            lines.remove(selectedRowBlueTable);

            // Update txt file
            Files.write(path, lines);
          } catch (IOException ex) {
            ex.printStackTrace();
          }
        }
```

Type: Code Block
Purpose: Delete a row from a table.and update deleted line in txt file
Description: If "DELETE_ DELETE_BLUE_TABLE_ROW" is equal to the "requestType" which came from the client, the code proceeds to execute the deletion of an admin entry. The code reads an integer value from the input using "reader.readLine()" and converts to integer type and it into the variable "selectedRowBlueTable". It then attempts to perform the deletion by accessing and manipulating a file named "admin.txt". The code reads all lines from the file into a list, removes the line at the index specified by "selectedRowBlueTable", and writes the updated lines back to the file.

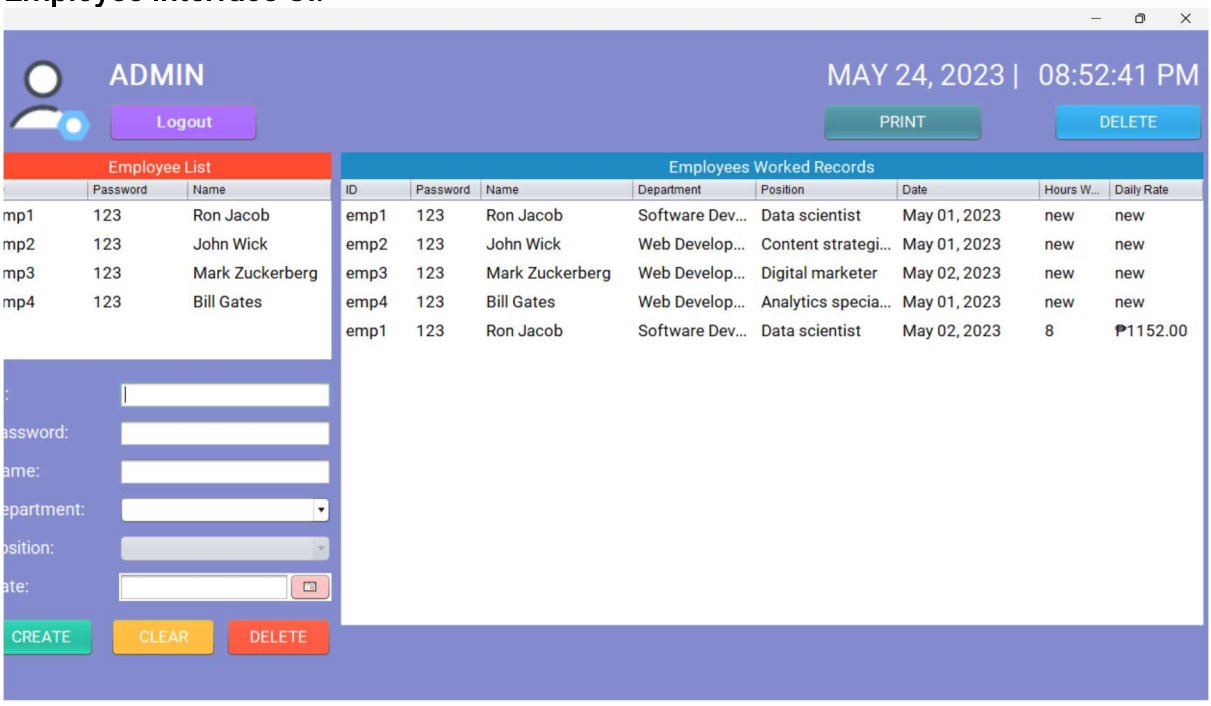
Code Snippet:

```
} else if ("DELETE_ORANGE_TABLE_ROW".equals(requestType)) {
          int selectedRowOrangeTable =
Integer.parseInt(reader.readLine());
          try {
            Path path = Paths.get("datas/employee_list.txt");
            List<String> lines = Files.readAllLines(path);

            String extractedLine = lines.get(selectedRowOrangeTable);
            String[] field = extractedLine.split("// ");

            lines.remove(selectedRowOrangeTable);

            Files.write(path, lines);

            File notepadFile = new File("employees/" + field[0] + ".txt");
            if (notepadFile.exists()) {
              notepadFile.delete();
            }
          } catch (IOException ex) {
            ex.printStackTrace();
          }
        }
```

Type: Code Block
Purpose: Delete an employee
Description: If " DELETE_ORANGE_TABLE_ROW_EMPLOYEE" is equal to "requestType", the code proceeds to execute the delete specific line from the personal employee txt file. The code reads an integer value from the input using "reader.readLine()"and assign into the variable "selectedRowOrangeTable". It then attempts to perform the delete the line by accessing and manipulating the file "employee_list.txt. The code reads all lines from the file into a list, removes the line at the index specified by "selectedRowOrangeTable", and writes the updated lines back to the file.

The employee ID field is extracted, and checks if the text file exists in the "employees" directory. If the text file exists, it is deleted.

Employee Interface UI:



EMPLOYEE INTERFACE CLIENT CODES:

Code Snippet:

```
    btnSubmit.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        Date selectedDate = dateChooser.getDate();

        if (selectedDate == null ||
cbHoursWorked.getSelectedItem() == null) {
          JOptionPane.showMessageDialog(null, "Please fill in
the blanks");
        } else {
          SimpleDateFormat dateFormat = new
SimpleDateFormat("MMM dd, yyyy");
          String formattedDate =
dateFormat.format(selectedDate);
          try (Socket socket = new Socket(SERVER_IP,
SERVER_PORT)) {
            PrintWriter writer = new
PrintWriter(socket.getOutputStream(), true);

            writer.println("SUBMIT");
            writer.println(formattedDate);
            writer.println(cbHoursWorked.getSelectedItem());
            writer.println(dailyRateDeduction((int)
cbHoursWorked.getSelectedItem()));

            Object[] row = new Object[3];
            row[0] = formattedDate;
            row[1] = cbHoursWorked.getSelectedItem();
            row[2] = "\u20B1" + dailyRateDeduction((int)
cbHoursWorked.getSelectedItem());
            model.addRow(row);

            JOptionPane.showMessageDialog(null, "Record
created successfully.");
            writer.close();
          } catch (IOException e1) {
            e1.printStackTrace();
          }
          cbHoursWorked.setSelectedIndex(-1);
        }
      }
    });
```

Type: Event Listener
Purpose: Submit daily worked record.
Description: When the "Submit" button is clicked. It validates the selected date and hours worked, and if they are not null, it sends the data to the server. The formatted date, hours worked, and daily rate deduction are sent to the server via a socket connection. The data is also added as a new row to the table model. Finally, a success message is displayed, and the hours worked selection is reset.

Code Snippet:

```
    btnDelete.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        int selectedRowTable = table.getSelectedRow();

        if (selectedRowTable >= 0) {
          int choice = JOptionPane.showConfirmDialog(null,
"Do you want to proceed?", "Confirmation",
              JOptionPane.YES_NO_OPTION);
          if (choice == JOptionPane.YES_OPTION) {
            try (Socket socket = new Socket(SERVER_IP,
SERVER_PORT)) {
              PrintWriter writer = new
PrintWriter(socket.getOutputStream(), true);

              writer.println("DELETE_EMPLOYEE_WORKED_RECORD");
              writer.println(selectedRowTable);

              model.removeRow(selectedRowTable);
              JOptionPane.showMessageDialog(null, "Deleted
successfully");
            } catch (IOException e1) {
              e1.printStackTrace();
            }
          }
        } else {
          JOptionPane.showMessageDialog(null, "Select a row
first");
        }
      }
    });
```

Type: Evet Listener
Purpose: Delete a selected row from the table.
Description: When clicked, it checks if a row is selected and prompts the user. for confirmation. If confirmed, it establishes a connection to the server and sends a request to delete the corresponding row in the table. The selected row is then removed from the table model, and a success message is displayed upon successful deletion. If no row is selected, a “Select a row first” is shown.

Code Snippet:

```
  public void loadEmployeeTable(DefaultTableModel model) {
    try (Socket socket = new Socket(SERVER_IP, SERVER_PORT))
{
      PrintWriter writer = new PrintWriter(new
OutputStreamWriter(socket.getOutputStream()), true);

      writer.println("EMPLOYEE_TABLE");
      BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
      String line;
      while ((line = reader.readLine()) != null) {
        String[] fields = line.split("// ");
        Object[] row = new Object[3];

        txtID.setText(fields[0]);
        txtPassword.setText(fields[1]);
        txtName.setText(fields[2]);
        txtDepartment.setText(fields[3]);
        txtPosition.setText(fields[4]);

        row[0] = fields[5]; // Date
        row[1] = fields[6]; // Hours Worked
        row[2] = fields[7]; // Daily Rate

        String regex = "\\d+";
        String text = fields[7];

        Pattern pt = Pattern.compile(regex);
        Matcher mt = pt.matcher(text);
        boolean result = mt.find();
        if (result) {
          row[2] = "\u20B1" + fields[7]; // Daily Rate
        } else {
          row[2] = fields[7]; // new
        }
        model.addRow(row);
      }
    } catch (IOException e1) {
      e1.printStackTrace();
    }
  }
```

Type: try-catch
Purpose: Display personal employee txt file
Description: The method loadEmployeeTable connects to a server using SERVER_IP and SERVER_PORT. It sends a request to update employee information and receives a response. The method processes the response line by line, extracting fields and updating text fields and a data model. Each line corresponds to an employee's details. The extracted fields populate specific text fields, while a new row is created and added to the data model. The row includes the date, hours worked, and rate

earned. If the daily rate consists only of digits, a peso sign is added. Otherwise, the original value is used. The method handles potential errors by printing stack traces.

Code Snippet:

```
public String dailyRateDeduction(double hoursWorked) {
  int hourlyRate = 0;
  String department = txtDepartment.getText();
  String pos = txtPosition.getText();
  // Based on industry trends and demand | Rate for each role
  if (department.equals("Software Development")) {
    if (pos.equals("Technical writer") || pos.equals("QA engineer")) {
      hourlyRate = 80;
    } else if (pos.equals("Front-end developer") || pos.equals("Back-end
developer")) {
      hourlyRate = 100;
    } else if (pos.equals("Database administrator") || pos.equals("Full-stack
developer")) {
      hourlyRate = 120;
    } else if (pos.equals("DevOps engineer") || pos.equals("Security
specialist")) {
      hourlyRate = 140;
    } else {
      hourlyRate = 160; // Security specialist
    }
  } else { // Web Development
    if (pos.equals("Technical writer") || pos.equals("Content strategist")) {
      hourlyRate = 70;
    } else if (pos.equals("UX/UI designer") || pos.equals("QA engineer")) {
      hourlyRate = 90;
    } else if (pos.equals("SEO specialist")) {
      hourlyRate = 110;
    } else if (pos.equals("Front-end developer") || pos.equals("Back-end
developer")) {
      hourlyRate = 130;
    } else if (pos.equals("Full-stack developer") || pos.equals("Analytics
specialist")) {
      hourlyRate = 150;
    } else {
      hourlyRate = 170; // DevOps engineer & Digital marketer
    }
  }
  double grossIncome = hoursWorked * hourlyRate;
  double taxRate = 0.1;

  double incomeTax = grossIncome * taxRate;
  double netIncome = grossIncome - incomeTax;
  String output = String.format("%.2f", netIncome);
  return output;
}
```

Type: Method
Purpose: Calculate the net income after tax deductions based on hours worked and job position.
Description: The dailyRateDeduction(double hoursWorked) method calculates the net income after tax deductions based on the hours worked and the job position.

Inside the method, it initializes the hourlyRate variable to 0 and retrieves the department and position values from the inputted txt fields (txtDepartment and txtPosition).

Based on the department and position values, the method determines the hourly rate for the job role. In the case of the "Software Development" department, specific

positions are assigned hourly rates ranging from 80 to 160. For the "Web Development" department, different positions have hourly rates ranging from 70 to 170.

The method calculates the gross income by multiplying the hours worked by the hourly rate and tax rate of 0.1 to calculate the income tax amount. The net income is obtained by subtracting the income tax from the gross income. The method format the net income as a string with two decimal places using the String.format() method and returns the formatted value. This method can be called to calculate the net income after tax deductions based on the hours worked and the job position.

EMPLOYEE INTERFACE SERVER CODES:

Code Snippet:

```
else if ("DELETE_EMPLOYEE_WORKED_RECORD".equals(requestType))
{
          int selectedRowTable =
Integer.parseInt(reader.readLine());
          try {
            BufferedReader reader1 = new BufferedReader(new
FileReader("datas/logged_in.txt"));
            String line = reader1.readLine();
            String[] fields = line.split("// ");
            String ID = fields[0];

            Path path = Paths.get("employees/" + ID +
".txt");
            List<String> lines = Files.readAllLines(path);

            lines.remove(selectedRowTable);

            Files.write(path, lines);
            reader1.close();
          } catch (IOException ex) {
            ex.printStackTrace();
          }
        }
```

Type: Delete Employee Worked Record
Purpose: Delete a specific employee's worked record from the file.
Description: This code snippet handles the deletion of an employee's worked record. It reads the selected row from the table, retrieves the employee's ID from the

"logged_in.txt" file, and constructs the file path. The lines from the file are read, and the selected row is removed. The updated lines are then written back to the file.

Code Snippet:

```
else if ("EMPLOYEE_TABLE".equals(requestType)) {
          sendEmployeeTableData(writerLogin);
        } ...

  private static void sendEmployeeTableData(PrintWriter
writer) {
    try (BufferedReader reader = new BufferedReader(new
FileReader("datas/logged_in.txt"))) {
      String line = reader.readLine();
      String[] fields = line.split("// ");
      String ID = fields[0];

      BufferedReader reader_1 = new BufferedReader(new
FileReader("employees/" + ID + ".txt"));
      String line_1;
      while ((line_1 = reader_1.readLine()) != null) {
        writer.println(line_1);
      }
      reader_1.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
```

Type: Send Employee Table Data
Purpose: Send the employee table data to the client.
Description: It reads who's currently logged in with "logged_in.txt" file, and the line from "logged_in.txt" file is split with "// " delimiter and store index 0 which corresponds to ID and use this to construct the file path for the employee's data file, and reads the lines from that file and each line is sent to the client using the PrintWriter.