Command Handler

Team members: Jad Abdallah (Application Server)

Jerome Chaker (Application Server)

Toufic Al Mabsout (Web Service)

Cynthia Ibrahim (Web Service)

Table of Contents

Abstract

This project involves the development of a server designed to manage and execute requests from an application server. The primary function of this server is to interact with a connected printer and perform various tasks based on commands received from the application server. The server is capable of handling a range of actions, including printing documents, sending information, receiving information, and showing notification pop-ups. Each action is processed efficiently by the web service, ensuring reliable communication between the application server and the printer.

Additionally, the system includes a user interface that facilitates direct user interactions. The user interface allows users to submit requests and commands through a web form, which are then transmitted to the server for processing. This interface enables users to specify print jobs, manage document transfers, and initiate other actions, providing a streamlined and intuitive way to interact with the server. By integrating user inputs with server-side processing, the system enhances the functionality of the printing infrastructure and ensures a seamless and responsive experience for managing print-related tasks.

Project Structure

Web Service

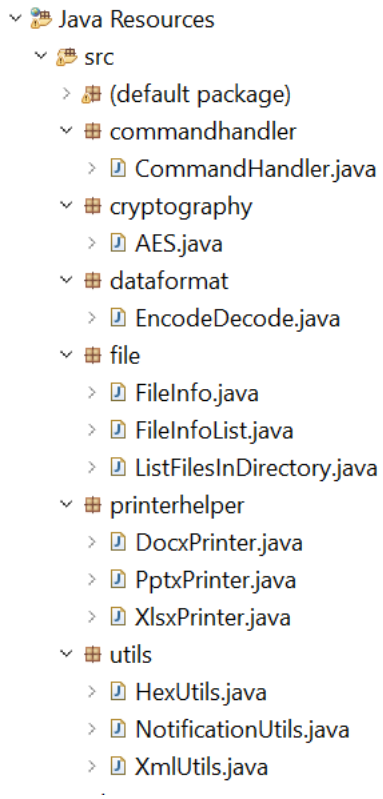

Figure 1 Project Web Service Structure

The main request is received in the CommandHandler.java file, other packages shown in the figure are used to provide methods and facilitate the functionalities to handle the main logic. The response is also sent from CommandHandler.java file.

Application Server

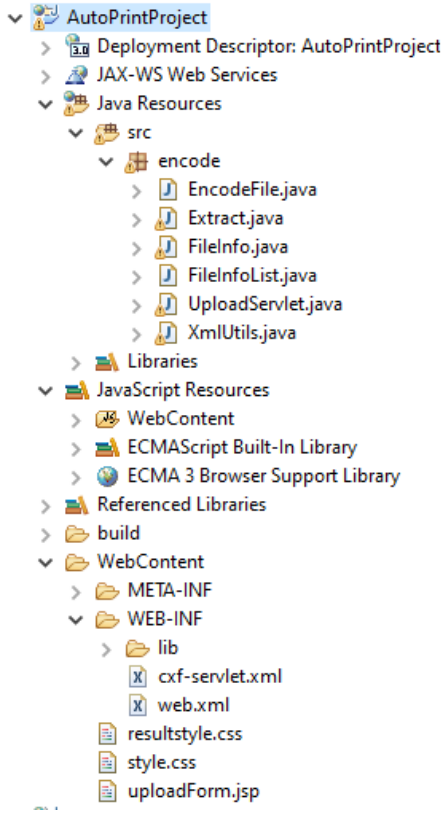

Figure 2 Project Application Server Structure

The main request is sent in the UploadServlet.java file, other packages shown in the figure are used to provide methods and facilitate the functionalities to handle the main logic, and EncodeFile.java writes the soap message. The response is also sent from UploadServlet.java file.

Utils

HexUtils.java

```
public class HexUtils {

    // Convert a byte array to a hexadecimal string
    public static String bytesToHex(byte[] bytes) {

    // Convert a hexadecimal string to a byte array
    public static byte[] hexStringToByteArray(String s) {
}
```

Figure 3 HexUtils.java

The HexUtils class provides utility methods for converting between byte arrays and hexadecimal strings. The bytesToHex method converts a byte array into its hexadecimal string representation, while the hexStringToByteArray method performs the reverse operation, converting a hexadecimal string into a byte array. The hexStringToByteArray method validates that the input string has an even length and contains valid hexadecimal characters, throwing an IllegalArgumentException if these conditions are not met. These methods are useful for encoding and decoding binary data in a readable hex format.

NotificationUtils.java

```
import javax.swing.*;

public class NotificationUtils {

    public static String popNotification(String message) {
}
```

Figure 4 NotificationUtils.java

The NotificationUtils class provides a utility method for displaying notifications to the user using a non-modal dialog. The popNotification method takes a string message as input and, if the message is not null or empty, creates and shows a dialog with the message centered and styled in a large font. The dialog includes an "OK" button to close it. The method ensures that the dialog is created and displayed on the Event Dispatch Thread for thread safety. If an error occurs during the dialog creation, it logs an error message. This class is useful for showing simple, user-friendly notifications within a Swing-based application.

XmlUtils.java

```
import com.fasterxml.jackson.databind.SerializationFeature;

public class XmlUtils {

    private static final XmlMapper xmlMapper = new XmlMapper();

    static {

    /**
     * Converts an ArrayList<FileInfo> to an XML string.
     *
     * @param fileInfos The ArrayList<FileInfo> to convert.
     * @return XML string representing the list.
     * @throws Exception If an error occurs during conversion.
     */
    public static String convertFileInfoListToXml(FileInfoList fileInfoList) throws Exception {
}
```

Figure 5 XmlUtils.java

The XmlUtils class provides functionality for converting data into XML format using the Jackson XmlMapper. The convertFileInfoListToXml method takes an instance of FileInfoList and converts it into an XML string. The class configures the XmlMapper to format the XML with indentation for better readability. This utility is useful for serializing FileInfoList objects into XML format for storage or communication purposes. The method throws an exception if any errors occur during the conversion process.

File

FileInfo.java

```
import javax.xml.bind.annotation.XmlElement;

@XmlRootElement(name = "FileInfo")
public class FileInfo {
    String fileName;
    String fileBase64;

     @XmlElement(name = "fileName")
    public String getFileName() {
        return fileName;
    }
    public void setFileName(String fileName) {
        this.fileName = fileName;
    }

    @XmlElement(name = "fileBase64")
    public String getFileBase64() {
        return fileBase64;
    }
    public void setFileBase64(String fileBase64) {
        this.fileBase64 = fileBase64;
    }
```

Figure 6 FileInfo.java

The FileInfo class represents a data model for storing file information, with fields for the file's name and its Base64-encoded content. It uses JAXB annotations to facilitate XML serialization and deserialization. The @XmlRootElement annotation specifies that this class maps to the root element of an XML document named "FileInfo." The @XmlElement annotations on the getter methods define the XML element names for the fileName and fileBase64 fields. This class is useful for encapsulating file data in a format suitable for XML-based communication or storage.

FileInfoList.java

```
import java.util.List;

@XmlRootElement(name = "FileInfoList")
public class FileInfoList {
    List<FileInfo> fileInfos;

     @XmlElement(name = "fileInfos")
    public List<FileInfo> getFileInfos() {
        return fileInfos;
    }

    public void setFileInfos(List<FileInfo> fileInfos) {
        this.fileInfos = fileInfos;
    }

}
```

Figure 7 FileInfoList.java

The FileInfoList class serves as a container for a list of FileInfo objects, representing a collection of file information. It is annotated with JAXB annotations to support XML serialization and deserialization. The @XmlRootElement annotation defines the root element of the XML document as "FileInfoList." The getFileInfos and setFileInfos methods, annotated with @XmlElement, manage the fileInfos property, which holds a list of FileInfo instances. This class facilitates the aggregation and XML representation of multiple FileInfo objects for data interchange or storage purposes.

ListFilesInDirectory.java

```java
import java.nio.file.DirectoryStream;

public class ListFilesInDirectory {

    /**
     * Returns a list of filenames in the specified directory.
     *
     * @param directoryPath the path of the directory to list files from
     * @return an ArrayList containing the names of the files in the directory
     */
    public static List<String> getFileNames(String directoryPath) {
}
```

Figure 8 ListFilesInDirectory.java

The ListFilesInDirectory class provides functionality for listing filenames within a specified directory. The getFileNames method accepts a directory path as a string and returns a list of filenames found in that directory. It checks if the provided path is a directory, then uses DirectoryStream to iterate over its contents. Only regular files (not subdirectories) are added to the result list. The method handles potential exceptions during directory traversal and returns an empty list if the path is not a directory or if an error occurs. This utility is useful for retrieving file names from a directory for further processing.

Dataformat

EncodeDecode.java

```
public class EncodeDecode {
    // Convert base64 string to bytes array
    public static byte[] decodeBase64(String base64String) {

    // Main function to encode in base64
    public static String encode(String filePath) {


    // Convert the file to a byte array
    public static byte[] readFileToByteArray(File file) throws IOException {

    // Convert from bytes to Base64
    public static String encodeToBase64(byte[] data) {

    // Save our bytes in a file path
    public static File saveBytesToFile(byte[] bytes, String filePath) throws IOException {
}
```

Figure 9 EncodeDecode.java

The EncodeDecode class provides utility methods for encoding and decoding data using Base64. The encode method reads a file specified by its path, converts its contents to a byte array, and then encodes that byte array to a Base64 string. The decodeBase64 method decodes a Base64-encoded string back into a byte array. Additional methods include readFileToByteArray, which reads a file into a byte array, encodeToBase64, which converts a byte array to a Base64 string, and saveBytesToFile, which writes a byte array to a file. These methods facilitate the conversion between binary data and Base64 encoding for file handling and data transfer.

Cryptography

AES.java

```
import javax.crypto.Cipher;

public class AES {
     // Defining the algorithm
     private static final String ALGORITHM = "AES/CBC/PKCS5Padding";

    // Function to Decrypt the password and decrypt the username and password
    public static boolean Decrypt(String username,String encPassword) throws Exception{

     // Decrypt from Hex and return it as Hex
     public static String decryptFromHex(String encryptedValueHex, SecretKeySpec key, IvParameterSpec iv) throws Exception {

    // Function to retrieve the properties from the given file path
    public static String[] getProperties(String filePath) {


}
```

Figure 10 AES.java

The AES class provides methods for decrypting data using AES encryption with CBC mode and PKCS5 padding. The Decrypt method validates a user's credentials by comparing provided and stored values. It retrieves encryption keys and data from a properties file, decrypts them using AES, and compares the decrypted values to the provided username and password. The decryptFromHex method handles the decryption of hexadecimal-encoded data. The getProperties method loads encryption parameters from a specified properties file. This class facilitates secure data decryption and user authentication in an application.

Interface

```jsp
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<!DOCTYPE html>
<html>
<head>
    <title>File Upload</title>
    <link rel="stylesheet" type="text/css" href="style.css">

</head>
<body>
    <div class="form-container">
        <h2>Upload File</h2>

        <form onload="uploadForm.jsp" action="uploadServlet" method="post" enctype="multipart/form-data">
            <div class="buttons">
                <label for="action">Choose action:</label>
                <select id="action" name="action" onchange="updateFormFields()">
                    <option value="print">Print</option>
                    <option value="transfer">Transfer</option>
                    <option value="retrieve">Retrieve</option>
                    <option value="message">Message</option>
                </select>
            </div>

            <label for="userName">User Name:</label>
            <input type="text" id="userName" name="userName" required>

            <label for="password">Password:</label>
            <input type="password" id="password" name="password" required>

             <label for="message" class="message-field hidden">Message:</label>
            <input type="text" id="message" name="message" class="message-field hidden">

            <label for="printerName" class="print-transfer-field hidden">Printer Name:</label>
            <input type="text" id="printerName" name="printerName" class="print-transfer-field hidden">

            <label for="fileRetrieve" class="retrieve-field hidden">File to Retrieve:</label>
            <input type="text" id="fileRetrieve" name="fileRetrieve" class="retrieve-field hidden">
```

Figure 11 interface code 1

```jsp
    <label for="outputPath" class="retrieve-field hidden">Path to Move File/Folder to:</label>
    <input type="text" id="outputPath" name="outputPath" class="retrieve-field hidden">

    <label for="flag" class="print-transfer-field hidden">Option:</label>
    <select id="flag" name="flag" class="print-transfer-field hidden">
        <option value="move">Move</option>
        <option value="delete">Delete</option>
        <option value="keep" selected>Keep</option>
    </select>

    <label for="file" class="print-transfer-field hidden">Select file:</label>
    <input type="file" id="file" name="file" class="print-transfer-field hidden">

    <label for="folderPath" class="print-transfer-field hidden">Folder Path:</label>
    <input type="text" id="folderPath" name="folderPath" class="print-transfer-field hidden">

  <%
        String error = request.getParameter("error");
        String empty = request.getParameter("empty");
        String error2= request.getParameter("error2");
        String error3= request.getParameter("error3");
        if ("true".equals(error)) { %>
            <p id="error-message" style="color: red;">Folder Path is required when command is 'Print'.</p>
        <% }
        if ("true".equals(empty)) { %>
            <p id="empty-message" style="color: red;">Message is required when command is 'Message'</p>
        <% }
        if ("true".equals(error2)) { %>
        <p id="error2-message" style="color: red;">Folder Path is required when command is 'Transfer'</p>
    <% }
        if ("true".equals(error3)) { %>
        <p id="error3-message" style="color: red;">Path to Retrieve is required when command is 'Retrieve'</p>
    <% }
    %>
```

Figure 12 interface code 2

```
            <input type="submit" value="Submit">
        </form>
    </div>
<script>
    function handleActionChange() {
        const action = document.getElementById('action').value;

        // Clear all messages when action changes
        const errorMessage = document.getElementById('error-message');
        const emptyMessage = document.getElementById('empty-message');
        const errorMessage2 = document.getElementById('error2-message');
        const errorMessage3 = document.getElementById('error3-message');

        if (errorMessage) errorMessage.style.display = 'none';
        if (emptyMessage) emptyMessage.style.display = 'none';
        if (errorMessage2) errorMessage2.style.display = 'none';
        if (errorMessage3) errorMessage3.style.display = 'none';

        // Update the URL with the selected action and remove error parameter
        const currentURL = new URL(window.location.href);
        currentURL.searchParams.set('action', action);
        currentURL.searchParams.delete('error');  // Remove the error parameter
        currentURL.searchParams.delete('empty');  // Remove the empty parameter
        currentURL.searchParams.delete('error2'); // Remove the error2 parameter
        currentURL.searchParams.delete('error3'); // Remove the error3 parameter

        window.history.pushState({}, '', currentURL);

        // Update form fields based on new action
        updateFormFields();
    }
```

Figure 13 interface code 3

```
    function updateFormFields() {
        const action = document.getElementById('action').value;

        // Show/Hide fields based on the selected action
        const printTransferFields = document.querySelectorAll('.print-transfer-field');
        const retrieveFields = document.querySelectorAll('.retrieve-field');
        const messageFields = document.querySelectorAll('.message-field');
        const outputPathLabel = document.querySelector('label[for="outputPath"]');

        // Initially hide all fields
        printTransferFields.forEach(field => field.classList.add('hidden'));
        retrieveFields.forEach(field => field.classList.add('hidden'));
        messageFields.forEach(field => field.classList.add('hidden'));

        // Show fields based on selected action
        if (action === 'retrieve') {
            retrieveFields.forEach(field => field.classList.remove('hidden'));
            outputPathLabel.textContent = 'Path to Retrieve:';
        } else if (action === 'print') {
            printTransferFields.forEach(field => field.classList.remove('hidden'));
            document.getElementById('outputPath').classList.remove('hidden');
            document.querySelector('label[for="outputPath"]').classList.remove('hidden');
            outputPathLabel.textContent = 'Path to Move File/Folder to:';
        } else if (action === 'transfer') {
            printTransferFields.forEach(field => {
                if (!field.classList.contains('hidden')) {
                    field.classList.remove('hidden');
                }
            });
```

Figure 14 interface code 4

```
            document.getElementById('outputPath').classList.remove('hidden');
            document.querySelector('label[for="outputPath"]').classList.remove('hidden');
            outputPathLabel.textContent = 'Path to Move File/Folder to:';
            document.getElementById('printerName').classList.add('hidden');
            document.getElementById('flag').classList.add('hidden');
            document.querySelector('label[for="printerName"]').classList.add('hidden');
            document.querySelector('label[for="flag"]').classList.add('hidden');
            document.getElementById('file').classList.remove('hidden');
            document.getElementById('folderPath').classList.remove('hidden');
            document.querySelector('label[for="file"]').classList.remove('hidden');
            document.querySelector('label[for="folderPath"]').classList.remove('hidden');
        } else if (action === 'message') {
            document.getElementById('message').classList.remove('hidden');
            document.querySelector('label[for="message"]').classList.remove('hidden');
        }
    }

    // Initialize form fields based on the default action
    window.onload = function() {
        const urlParams = new URLSearchParams(window.location.search);
        const action = urlParams.get('action') || 'print'; // Default to 'print' if not set
        document.getElementById('action').value = action;
        updateFormFields(); // Set the initial state of the form fields
        document.getElementById('action').addEventListener('change', handleActionChange);
    };
</script>


</body>
</html>
```

Figure 15 interface code 5

This JSP page is designed to dynamically handle different file upload scenarios based on user-selected actions such as "Print," "Transfer," "Retrieve," or "Message." It features a form where the available fields change according to the selected action, showing or hiding relevant inputs for user name, password, file selection, and additional parameters. Error messages are conditionally displayed if required fields are missing or incorrect, based on query parameters. JavaScript manages form field visibility and updates the URL to reflect the selected action, while handling initialization and field updates on page load and user interaction.

Methods

```
package encode;

import java.io.FileInputStream;

public class Extract {

    //new instance of the encoding class
     EncodeFile enc = new EncodeFile();

     public String saveFileAndGetPath(Part filePart, String folderPath) throws IOException {

     //Method that returns the string Base64 of a file by calling the encode class method and giving it a filepath parameter
        public String saveFileAndEncode(String filePath) throws IOException {

            // method to delete file from the filepath
       public void deleteFile(String filePath) {

       //method to extract the filename from the browse option in the interface
        public String extractFileNamePart(Part filePart) {

        //method to extract the filename from the filepath
        public String extractFileName(Path filePath) {

        //method that goes through a folder using the folderpath, and fills an array with all the filepaths in this folder
        public ArrayList<Path> listFilesInDirectory(String directoryPath) throws IOException {

        //method to move transferred files to a folder
        public void moveFile(String sourceFilePath) throws IOException {


        public static String getLogPath(String filePath) {
```

Figure 16 methods used

The Extract class provides functionality for handling file operations, including saving, encoding, and deleting files. It uses an EncodeFile instance to encode files into Base64 format. The class includes methods to save a file to a specified path, encode it, and optionally delete it afterward. It also extracts filenames from file parts and file paths, lists all files in a directory, and moves files to a specific directory. Additionally, it can retrieve a logging path from a properties file. The class handles various file-related tasks and exceptions, ensuring smooth file management and operations.

Encoding

```java
package encode;

import javax.jws.WebMethod;

@XmlRootElement(name = "sendFilePath", namespace = "http://encode/")
@WebService
    public class EncodeFile {

    //Method to read the encoded file and return it as a String
         public String encode(String filepath) {

        //method to read all bytes in a file;
          public byte[] readFileToByteArray(File file) throws IOException {

        //method to convert from bytes to base64
          public String encodeToBase64(byte[] data) {

        //Password encryption
        private static final String ALGORITHM = "AES/CBC/PKCS5Padding";
        private static final String STATIC_IV_HEX =  "C:/Users/User2/Desktop/info.properties";

        // Convert a hexadecimal string to a byte array
        public static byte[] hexStringToByteArray(String s) {

        // Convert a byte array to a hexadecimal string
        public static String byteArrayToHexString(byte[] bytes) {

        // Encrypt the password using AES-256 with CBC mode and static IV
        public static String encryptPassword(String password, String keyHex) throws GeneralSecurityException, IOException {

        // Load the AES key from the properties file
        public static String loadKeyFromProperties(String propertyfile) throws IOException {

        //Load the IV key from the properties file
        public static String loadivFromProperties() throws IOException {
```


Figure 17 encoding methods

The EncodeFile class provides methods for encoding files to Base64 and encrypting passwords using AES encryption. It includes functionality to read a file into a byte array, convert that byte array to a Base64-encoded string, and methods to encrypt passwords using AES with a static initialization vector (IV) read from a properties file. The class uses the AES/CBC/PKCS5Padding algorithm for encryption, where it converts hexadecimal keys and IVs to byte arrays for the cipher. It also contains utility methods to handle hexadecimal string conversions and to load encryption keys and IVs from a properties file.

Soap request

```
//Soap request operation
    @WebMethod(operationName= "sendFilePath")
    public String sendFilePath(FileInfoList fileInfoList, String command, String printerName, String outputPath,
            String flag,String fileRetrieve,String message, String userName, String password){
          try {
                // Create a SOAP message and fill it with the file path
                SOAPConnectionFactory soapConnectionFactory = SOAPConnectionFactory.newInstance();
                SOAPConnection soapConnection = soapConnectionFactory.createConnection();


                MessageFactory messageFactory = MessageFactory.newInstance();
                SOAPMessage soapMessage = messageFactory.createMessage();

               SOAPPart soapPart = soapMessage.getSOAPPart();
               SOAPEnvelope envelope = soapPart.getEnvelope();
               envelope.addNamespaceDeclaration("ns", "http://encode/");

               SOAPBody soapBody = envelope.getBody();
                SOAPBodyElement bodyElement = (SOAPBodyElement) soapBody.addChildElement("sendFilePath", "ns", "http://encode/");

                //converted Array into Xml String
                String fileInfosXml = XmlUtils.convertFileInfoListToXml(fileInfoList);
                bodyElement.addChildElement("arg0").addTextNode(fileInfosXml);

                //Adding the other String elements to the message
                bodyElement.addChildElement("arg1").addTextNode(command);
                bodyElement.addChildElement("arg2").addTextNode(printerName);
                bodyElement.addChildElement("arg3").addTextNode(outputPath);
                bodyElement.addChildElement("arg4").addTextNode(flag);
                bodyElement.addChildElement("arg5").addTextNode(fileRetrieve);
                bodyElement.addChildElement("arg6").addTextNode(message);
                bodyElement.addChildElement("arg7").addTextNode(userName);
                bodyElement.addChildElement("arg8").addTextNode(password);
```

Figure 18 request 1

```
                soapMessage.saveChanges();

                // Request SOAP message
                System.out.println("Request SOAP Message:");
                soapMessage.writeTo(System.out);
                System.out.println();

                // Endpoint of the other server
                URL endpointURL = new URL("http://192.168.0.109:8080/CommandProject/services/commandhandler");
                // Send request and receive the base64 string of the content
                SOAPMessage response = soapConnection.call(soapMessage, endpointURL);

                // Access the SOAP Body
                SOAPBody body = response.getSOAPBody();

            // Retrieve the return tag's value
        SOAPElement sendFilePathResponse = (SOAPElement) body.getChildElements(new QName("http://encode/", "sendFilePathResponse")).next();
        SOAPElement returnElement = (SOAPElement) sendFilePathResponse.getChildElements(new QName("return")).next();
           String returnValue = returnElement.getValue();

               // Print the value of the 64base string
                System.out.println("Return Value: " + returnValue);

                    return returnValue;

                } catch (Exception e) {
                    e.printStackTrace();
                    return "Error occurred: " + e.getMessage();
                }

        }
}
```


Figure 19 request 2

The sendFilePath method is a SOAP web service operation that constructs and sends a SOAP request message to a specified endpoint. It creates a SOAP message using the SOAPConnectionFactory and MessageFactory classes, adding necessary elements to the SOAP body, including a serialized XML representation of a FileInfoList object and several string parameters such as command, printer name, output path, and user credentials. After assembling the SOAP message, it is sent to a remote service using the SOAPConnection object. The response is processed to extract and return a Base64-encoded string from the SOAP message body. The method handles exceptions by printing the stack trace and returning an error message if an issue occurs.

Servlet handling

```
package encode;

import javax.servlet.ServletException;

@MultipartConfig
public class UploadServlet extends HttpServlet {

    //instances of classes initialization
    Extract ext = new Extract();
    EncodeFile enc = new EncodeFile();

    private static final Logger logger = Logger.getLogger(UploadServlet.class.getName());
    static {

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        //retrieving data from the interface
         String folderPath = request.getParameter("folderPath");
         List<FileInfo> fileInfos = new ArrayList<>();

         Part filePart = request.getPart("file");
          String command = request.getParameter("action");
          String printerName = request.getParameter("printerName");
          String message= "";
          String outputPath = request.getParameter("outputPath");
          String flag = request.getParameter("flag");
          String fileRetrieve="";
          String userName = request.getParameter("userName");
          String nonencryptedpassword = request.getParameter("password");
          String PROPERTIES_FILE =  "C:/Users/User2/Desktop/info.properties";
          String key= EncodeFile.loadKeyFromProperties(PROPERTIES_FILE);
```

Figure 20 servlet 1

```
 //check if the folder path is empty, so we are working with a file instead of a folder
if ( filePart== null || filePart.getSize() == 0) {
    try {
        //System.out.println(folderPath);
        ArrayList<Path> filePaths = ext.listFilesInDirectory(folderPath);
        //System.out.println(filePaths.get(0));

        for (Path filePath : filePaths) {

            String fileName = ext.extractFileName(filePath);

            FileInfo fileInfo = new FileInfo();
            fileInfo.setFileBase64(filePath.toString());
            fileInfo.setFileName(fileName);

            fileInfos.add(fileInfo);
        }
           int i=0;
        for (FileInfo fileInfo : fileInfos) {
            String base64EncodedFile = ext.saveFileAndEncode(fileInfo.getFileBase64());
            FileInfo fileInf=new FileInfo();
            fileInf.setFileName(fileInfo.getFileName());
            fileInf.setFileBase64(base64EncodedFile);
            fileInfos.set(i,fileInf);
            i=i+1;
            logger.info(command+" "+flag+" "+fileInfo.getFileName());

        }
        //if we are transferring the files, then i will send it to a specific folder
        if ("transfer".equals(command)) {
            for (Path filePath : filePaths){
                ext.moveFile(filePath.toString());
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
```

Figure 21 servlet 2

```
            //else we are working with a folder
        }else{

try{
                String oneFilePath=ext.saveFileAndGetPath(filePart,folderPath);
                Path path= Paths.get(oneFilePath);
                  ArrayList<Path> filePaths = new ArrayList<Path>();
                  filePaths.add(path);

                  for (Path filePath : filePaths) {

                      String fileName = ext.extractFileName(filePath);

                      FileInfo fileInfo = new FileInfo();
                      fileInfo.setFileBase64(filePath.toString());
                      fileInfo.setFileName(fileName);

                      fileInfos.add(fileInfo);
                  }
                  System.out.println(filePaths.size());
                  int i=0;
                  for (FileInfo fileInfo : fileInfos) {
                      String base64EncodedFile = ext.saveFileAndEncode(fileInfo.getFileBase64());
                      FileInfo fileInf=new FileInfo();
                      fileInf.setFileName(fileInfo.getFileName());
                      fileInf.setFileBase64(base64EncodedFile);
                      fileInfos.set(i,fileInf);
                      i=i+1;
                      logger.info(command+" "+flag+" "+fileInfo.getFileName());

                  }
              } catch (IOException e) {
                  e.printStackTrace();
              }
        }
```

Figure 22 servlet 3

```
String password="";
FileInfoList fileInfoList = new FileInfoList();
fileInfoList.setFileInfos(fileInfos);
try {
    password = EncodeFile.encryptPassword(nonencryptedpassword, key);
} catch (GeneralSecurityException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
}

if ("retrieve".equals(command)) {

     if (command.equals("retrieve") && (outputPath == null || outputPath.trim().isEmpty())) {
        // Handle the error case where folderPath is required
         response.sendRedirect("uploadForm.jsp?action=retrieve&error3=true");
        return; // Exit the method to prevent further processing
    }

     fileRetrieve=request.getParameter("fileRetrieve");

   String fileInfosXml = enc.sendFilePath(fileInfoList, command, printerName,outputPath, flag,fileRetrieve,message, userName, password);

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html><head>");
    out.println("<link rel='stylesheet' type='text/css' href='resultstyle.css'>");
    out.println("</head><body>");
    out.println("<div class='container'>");

    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder builder = null;
    try {
        builder = factory.newDocumentBuilder();
    } catch (ParserConfigurationException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    }
```


Figure 23 servlet 4

```
    Document doc = null;
    try {
        doc = builder.parse(new InputSource(new StringReader(fileInfosXml)));
    } catch (SAXException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    }

    // Get all <fileInfos> elements
    NodeList fileInfosList = doc.getElementsByTagName("fileInfos");
    //System.out.println(fileInfosList.getLength());

    try{

         for (int i = 1; i < fileInfosList.getLength(); i++) {
             Element fileInfo = (Element) fileInfosList.item(i);
             String fileBase64 = fileInfo.getElementsByTagName("fileBase64").item(0).getTextContent();
             String fileName = fileInfo.getElementsByTagName("fileName").item(0).getTextContent();
            // System.out.println("fileBase64: " + fileBase64);
               logger.info(command+" "+flag+" "+fileName);

    byte[] bytes= Base64.getDecoder().decode(fileBase64);

    File file = new File("C:/Users/User2/Desktop/RetrievedFile/"+fileName);
    try (FileOutputStream foutput = new FileOutputStream(file)) {
        foutput.write(bytes);
        out.println("<h2>Success: File" + fileName +" created successfully at " + file.getAbsolutePath() + ".</h2>");
        logger.info("File retrieved");
```

Figure 24 servlet 5

```
            } catch (IOException e) {
                out.println("<h2>Error: Failed to write file. " + e.getMessage() + "</h2>");
            }catch (IllegalArgumentException e){
                out.println("<h2>Error: Failed to decode Base64 string. " + e.getMessage() + "</h2>");
            }
        }
                 out.println("<input type='button' value='Return' onclick='window.location.href=\"uploadForm.jsp?action=retrieve\"'> ");
                 out.println("</div>");
                 out.println("</body></html>");

            }finally{}
        }
        else if(command.equals("message")){
                message=request.getParameter("message");
         if (command.equals("message") && (message == null || message.trim().isEmpty())) {
                // Handle the error case where folderPath is required
             response.sendRedirect("uploadForm.jsp?action=message&empty=true");
                return; // Exit the method to prevent further processing
            }
        String msgresult = enc.sendFilePath(fileInfoList, command, printerName,outputPath, flag, fileRetrieve,message, userName, password);
        logger.info(msgresult);

    response.setContentType("text/html");
    response.getWriter().println("<html><head>");
    response.getWriter().println("<link rel='stylesheet' type='text/css' href='resultstyle.css'>");
    response.getWriter().println("</head><body>");
    response.getWriter().println("<div class='container'>");
    response.getWriter().println("<h2> "+ msgresult +"</h2>");
    response.getWriter().println("<input type='button' value='Return' onclick='window.location.href=\"uploadForm.jsp?action=message\"'> ");
    response.getWriter().println("</div>");
    response.getWriter().println("</body></html>");
       }
```

Figure 25 servlet 6

```
                else{
                    if (command.equals("print") && (folderPath == null || folderPath.trim().isEmpty())) {
                       // Handle the error case where folderPath is required
                        response.sendRedirect("uploadForm.jsp?error=true");
                       return; // Exit the method to prevent further processing
                    }
                     if (command.equals("transfer") && (folderPath == null || folderPath.trim().isEmpty())) {
                       // Handle the error case where folderPath is required
                        response.sendRedirect("uploadForm.jsp?action=transfer&error2=true");
                       return; // Exit the method to prevent further processing
                    }
        String result = enc.sendFilePath(fileInfoList, command, printerName,outputPath, flag, fileRetrieve,message, userName, password);
        logger.info(result);
        // Send response
        response.setContentType("text/html");
        response.getWriter().println("<html><head>");
        response.getWriter().println("<link rel='stylesheet' type='text/css' href='resultstyle.css'>");
        response.getWriter().println("</head><body>");
        response.getWriter().println("<div class='container'>");
        response.getWriter().println("<h2>Result: " + result + "</h2>");
        if (command.equals("transfer")){
        response.getWriter().println("<input type='button' value='Return' onclick='window.location.href=\"uploadForm.jsp?action=transfer\"'>
        }
        else{
            response.getWriter().println("<input type='button' value='Return' onclick='window.location.href=\"uploadForm.jsp\"'> ");
        }
        response.getWriter().println("</div>");
        response.getWriter().println("</body></html>");
    }

}
}
```


Figure 26 servlet 7

The UploadServlet class handles file uploads and operations based on user commands in a web application. It starts by initializing logging and retrieving input parameters from the HTTP request, including file parts, folder paths, and user credentials. The servlet distinguishes between handling individual file uploads and processing files from a directory. For file uploads, it encodes files in Base64, optionally moves them based on the command, and logs the operations. Commands are processed as follows: retrieve command fetches files from a service, decodes them from Base64, saves them to a specified directory, and displays success or error messages; message command sends a message to a service and shows the result; print and transfer commands handle printing and transferring files respectively, with error handling for missing parameters. The servlet generates HTML responses with results and provides options to return to previous pages based on the command.

Printerhelper

DocxPrinter.java

```
public class DocxPrinter {

    public static void main(String[] args) {

    public void printDocx(String filePath, String printerName) throws IOException, PrinterException {

    private PrintService getPrintService(String printerName) {

    static class DocxPrintable implements Printable {
}
```

Figure 27 DocxPrinter.java

The DocxPrinter class facilitates printing .docx documents using the Java Print API. The printDocx method loads a DOCX file, sets up a PrinterJob, and configures it to use a DocxPrintable implementation to render the document. The getPrintService method retrieves the appropriate PrintService, either the default or one specified by name. The DocxPrintable inner class implements the Printable interface to handle the rendering of document paragraphs onto the printed page. The class supports sending documents to a printer and handles cases where the specified printer is not found.

PptxPrinter.java

```
import org.apache.poi.xslf.usermodel.XMLSlideShow;

public class PptxPrinter {

    public static void main(String[] args) {

    public void printPptx(String filePath, String printerName) throws IOException, PrinterException {

    private PrintService getPrintService(String printerName) {

    static class PptxTextPrintable implements Printable {
}
```

Figure 28 PptxPrinter.java

The PptxPrinter class provides functionality to print .pptx (PowerPoint) slides using the Java Print API. The printPptx method loads a PPTX file, retrieves the first slide, and sets up a PrinterJob with a custom PptxTextPrintable implementation to handle rendering. The getPrintService method locates the appropriate PrintService, either default or specified by name. The PptxTextPrintable inner class implements the Printable interface to render slide text onto the printed page, handling font properties and positioning for text shapes. This class supports sending slides to a printer and manages cases where the specified printer is not found.

XlsxPrinter.java

```
import org.apache.poi.ss.usermodel.*;

public class XlsxPrinter {

    public static void main(String[] args) {

    public void printXlsx(String filePath, String printerName) throws IOException, PrinterExceptio

    private PrintService getPrintService(String printerName) {

    static class XlsxPrintable implements Printable {
}
```

Figure 29 XlsxPrinter.java

The XlsxPrinter class enables printing .xlsx (Excel) spreadsheets through the Java Print API. The printXlsx method loads an Excel file, retrieves the first sheet, and sets up a PrinterJob with a custom XlsxPrintable implementation for rendering. The getPrintService method locates the specified or default PrintService. The XlsxPrintable inner class implements the Printable interface to render spreadsheet content, drawing rows and cells with appropriate borders and adjusting column widths based on content. This class manages the printing of Excel documents and handles cases where the specified printer is not found.

Command handler

```
@WebService(name="EncodeFile",targetNamespace="http://encode/")
public class CommandHandler {

    public static final Logger logger = Logger.getLogger(CommandHandler.class.getName());

    static {
       try {
           // Create a file handler that writes log messages to a file
           String propertyPath = "C:\\Users\\E14\\Desktop\\Properties\\info.properties";
           String logFilePath = getLogPath(propertyPath);
           FileHandler fileHandler = new FileHandler(logFilePath,10*1024,1, true); // Append mode
           fileHandler.setFormatter(new SimpleFormatter()); // Use a simple text format
           logger.addHandler(fileHandler);
       } catch (IOException e) {
           e.printStackTrace();
       }
   }
```

Figure 30 Logging

The CommandHandler class includes a static logger instance configured to log messages to a file. The logger is initialized in a static block, which sets up a FileHandler to write logs to a file specified by the getLogPath method, located in C:\Users\E14\Desktop\Properties\info.properties. The FileHandler is set to append mode with a maximum file size of 10 KB and a single backup file. The logs are formatted using a SimpleFormatter. This configuration ensures that log messages are recorded persistently and in a readable format.

```
@WebMethod(operationName = "sendFilePath")
 public String receivePath(String fileInfosXml, String command, String printerName,String directory, String flag,String f
    logger.info("\n\n\n");
    logger.info("command: "+command);
    logger.info("printer name: "+printerName);
    logger.info("directory: "+directory);
    logger.info("flag :"+flag);
    logger.info("file to retrieve: "+fileRetrieve);
    logger.info("message: "+message);
    logger.info("username: "+username);
    logger.info("encrypted password: "+encPassword);

    boolean validCredentials = ValidateCredentials(username,encPassword);
    if (validCredentials){

        // Handle retrieve case
        if (command.equals("retrieve")){
            return retrieveFile(command,directory,fileRetrieve);
        }
        // Handle notification case
        else if (command.equals("message")){
            return popNotification(message);
        }
        // Handle Print or Transfer since they are similar
        else{

        // System.out.println(fileInfosXml);
```

Figure 31 Function receivePath 1

```
// Parse the XML
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse(new InputSource(new StringReader(fileInfosXml)));

// Get all <fileInfos> elements
NodeList fileInfosList = doc.getElementsByTagName("fileInfos");
for (int i = 1; i < fileInfosList.getLength(); i++) {
    Element fileInfo = (Element) fileInfosList.item(i);
    String fileBase64 = fileInfo.getElementsByTagName("fileBase64").item(0).getTextContent();
    String fileName = fileInfo.getElementsByTagName("fileName").item(0).getTextContent();
    // System.out.println("fileBase64: " + fileBase64);
    // System.out.println("fileName: " + fileName);
    // System.out.println("---");
 if (ValidateExtension(fileName)){
try {
    String filePath = (directory.isEmpty()) ? "C:\\Users\\E14\\Desktop\\TestingPrint\\"+fileName : directory+"\\"+fileName;
    // Retrieve command requires the argument fileRetrieve

    //Generate Directory if it does not exist
    File file = new File(filePath);
    File parentDir = file.getParentFile();

    if (parentDir != null && !parentDir.exists()) {
        if (parentDir.mkdirs()) {
            logger.info("directory created: "+parentDir.getAbsolutePath());
            // System.out.println("Directory created: " + parentDir.getAbsolutePath());
        } else {
            // System.out.println("Failed to create directory: " + parentDir.getAbsolutePath());
            logger.info("failed to create directory");

        }
    }
```

Figure 32 Function receivePath 2

```
 // Convert the base64 string to byte array
byte[] responseBytes = EncodeDecode.decodeBase64(fileBase64);
// Save the content of the received content into a file
EncodeDecode.saveBytesToFile(responseBytes, filePath);



// System.out.println(fileName+" "+command+" "+" "+fileBase64);
logger.info("file name: "+fileName);
// Perform the print command or other operations as needed
if (command.equals("print")){
    if (filePath.endsWith(".pdf") || filePath.endsWith(".txt") || filePath.endsWith(".csv")||
        filePath.endsWith(".html") || filePath.endsWith(".htm")){
          printFileDefault(filePath, printerName,flag);
    }
    else if (filePath.endsWith(".docx")){
        DocxPrinter docxPrinter = new DocxPrinter();
        docxPrinter.printDocx(filePath, printerName);
    }
    else if (filePath.endsWith(".xlsx")){
        XlsxPrinter xlsxPrinter = new XlsxPrinter();
        xlsxPrinter.printXlsx(filePath, printerName);
    }
    else if (filePath.endsWith(".pptx")){
        PptxPrinter pptxPrinter = new PptxPrinter();
        pptxPrinter.printPptx(filePath, printerName);
    }

    }


else if (command.equals("transfer")){
    logger.info("file transfered");
}
```

Figure 33 Function receivePath 3

```
            else{
                logger.info("unavailable command");
                return "unavailable command";
            }


        }catch (Exception e) {
             e.printStackTrace();
             logger.info("Error occurred: " + e.getMessage());
             return "Error occurred: " + e.getMessage();
          }
    }
         else {
             logger.info("invalid extension for file: "+fileName);
          }
         }
    return command + " done successfully";
      }
    }
else{
    logger.info("Invalid credentials, unable to perform the requested action.");
    return "Invalid credentials, unable to perform the requested action.";
}
```

Figure 34 Function receivePath 4

The receivePath method is a web service operation designed to handle various commands related to file operations, including file retrieval, printing, and notifications. The method logs the received parameters and validates user credentials before proceeding with the requested action.

1. **Parameter Logging**: The method logs detailed information about the command, printer name, directory, flag, file retrieval request, message, username, and encrypted password.
2. **Credential Validation**: It checks the validity of the provided credentials using the ValidateCredentials method. If the credentials are invalid, it logs an error message and returns a failure response.
3. **Command Handling**:
    - **Retrieve**: If the command is "retrieve", it calls the retrieveFile method to handle file retrieval from the specified directory.
    - **Message**: If the command is "message", it invokes the popNotification method to display the provided message.
    - **Print/Transfer**: For "print" and "transfer" commands, it processes the provided fileInfosXml, parsing it to extract file information. The XML is parsed to retrieve file base64-encoded data and file names. It then decodes the base64 data, saves the file, and performs the requested action:

- **Print**: The method determines the file type and invokes the appropriate printer class (DocxPrinter, XlsxPrinter, or PptxPrinter) or a default print method based on the file extension.
- **Transfer**: Logs a message indicating that the file transfer was successful.

4. **Exception Handling**: If any exceptions occur during file processing or printing, they are caught, logged, and a corresponding error message is returned.
5. **Directory Creation**: If necessary, it creates directories to ensure the file can be saved properly.

The method returns a confirmation message indicating the success of the command or an error message if the operation fails.

```
// Function that retrieves the file/folder
 public String retrieveFile(String command, String directory, String fileRetrieve) throws Exception {

        // Create the List of FileInfo and store this List in an object called retrieveList that has
        // As attribute a List of FileInfo
        FileInfoList retrieveList = new FileInfoList();
        List<FileInfo> retrieve = new ArrayList<>();

        // If fileRetrieve is Empty, then handle the case when the user wants a folder
        if (fileRetrieve.isEmpty()){

            // Function to retrieve all file names in the given directory
            List<String> fileNames = ListFilesInDirectory.getFileNames(directory);

            // Loop over the fileNames, generate the path and then appply base64 encoding and store
            // The enconding with the name of the file in the object FileInfo and add it to the list
            for (String fileName : fileNames){
                String filePath = (directory.isEmpty()) ? "C:\\Users\\E14\\Desktop\\TestingPrint\\"+fileName : directory+"\\"+fileName;
                logger.info("file name: "+fileName+"\ncommand: "+command);
                // System.out.println(filePath);
                String encodeBase64 = EncodeDecode.encode(filePath);
                logger.info("file "+filePath+" retrieved");
                Path path = Paths.get(filePath);
                String fileNameRetrieve = path.getFileName().toString();
                FileInfo fileInfoRetrieve = new FileInfo();
                fileInfoRetrieve.setFileBase64(encodeBase64);fileInfoRetrieve.setFileName(fileNameRetrieve);
                if (ValidateExtension(fileNameRetrieve)) retrieve.add(fileInfoRetrieve);
                else logger.info("invalid extension for file: "+fileNameRetrieve);

            }
```

Figure 35 Function retrieveFile 1

```
            // Finally set the list and convert it to Xml then return it
            retrieveList.setFileInfos(retrieve);
            String fileInfosXmlRetrieve = XmlUtils.convertFileInfoListToXml(retrieveList);
            return fileInfosXmlRetrieve;
        }

        // Same case but taking one file which is fileRetrieve from the command
        else{

            String filePath = (directory.isEmpty()) ? "C:\\Users\\E14\\Desktop\\TestingPrint\\"+fileRetrieve : directory+"\\"+fileRetrieve;
            logger.info("file name: "+fileRetrieve+"\ncommand: "+command);
            // System.out.println(filePath);
            String encodeBase64 = EncodeDecode.encode(filePath);
            logger.info("file "+filePath+" retrieved");
            Path path = Paths.get(filePath);
             String fileNameRetrieve = path.getFileName().toString();
            FileInfo fileInfoRetrieve = new FileInfo();
            fileInfoRetrieve.setFileBase64(encodeBase64);fileInfoRetrieve.setFileName(fileNameRetrieve);
            if (ValidateExtension(fileNameRetrieve)) retrieve.add(fileInfoRetrieve);
            else logger.info("invalid extension for file: "+fileNameRetrieve);
            retrieveList.setFileInfos(retrieve);
            String fileInfosXmlRetrieve = XmlUtils.convertFileInfoListToXml(retrieveList);
            return fileInfosXmlRetrieve;
        }

    }
```

Figure 36 Function retrieveFile 2

The retrieveFile method is responsible for retrieving files from a specified directory or a single file based on the provided command. It encodes the file content in Base64, organizes the file information, and returns it as an XML string.

Parameters:

- **command**: A string indicating the type of command. Used to determine if the retrieval is for a single file or a folder of files.
- **directory**: A string representing the directory from which files are to be retrieved.
- **fileRetrieve**: A string specifying the name of the file to retrieve. If empty, the method retrieves all files in the specified directory.

Return Value:

- Returns an XML string containing Base64-encoded file data and file names. If multiple files are retrieved, they are included in a list format.

Functionality:

1. **Initialization**:
    - Initializes a FileInfoList object and a list of FileInfo objects to store information about retrieved files.

2. **Directory Retrieval**:
    - If fileRetrieve is empty, indicating that a directory listing is requested:
        - Retrieves all file names from the specified directory using the ListFilesInDirectory.getFileNames method.
        - Iterates over the list of file names, constructs the file path, and encodes the file content in Base64 using the EncodeDecode.encode method.
        - Creates FileInfo objects for each file, setting the encoded content and file name. Only files with valid extensions (as determined by ValidateExtension) are added to the list.
        - Converts the FileInfoList to XML format using XmlUtils.convertFileInfoListToXml and returns the XML string.
3. **Single File Retrieval**:
    - If fileRetrieve is not empty, indicating a single file retrieval:
        - Constructs the file path from the directory and file name.
        - Encodes the file content in Base64 and creates a FileInfo object with the encoded content and file name.
        - Adds the FileInfo object to the list if the file extension is valid.
        - Converts the FileInfoList to XML format and returns the XML string.

Error Handling:

- Logs detailed information about each file retrieval process.
- Logs a warning if a file with an invalid extension is encountered.

Dependencies:

- EncodeDecode.encode: Encodes file content to Base64.
- ValidateExtension: Checks if the file extension is valid.
- XmlUtils.convertFileInfoListToXml: Converts FileInfoList to XML format.
- ListFilesInDirectory.getFileNames: Retrieves file names from a directory.

```
    // Prints the file and return to the user the what happened
    public String printFileDefault(String filePath, String printerName,String flag) {

    // Transfer the file to the same filePath but in an archive folder
    public void transferFileToArchive(String filePath) {

    // Delete the file
    public static void deleteFile(String filePath) {

    // Function that calls AES class and validates user credentials
    public boolean ValidateCredentials(String username, String encPassword) throws Exception{
        return AES.Decrypt(username, encPassword);
    }

    // Function to retrieve the properties from the given file path
    public static String getLogPath(String filePath) {

    // Function to store allowed extensions in a List
    public static List<String> getAllowedExtensions(String filePath) {

    // Function to validate extensions
    public boolean ValidateExtension(String fileName){

    // Helper method to get file extension with the dot
   public static String getFileExtension(String fileName) {

    // Method to display the notification
    public String popNotification(String message) {
}
```

Figure 37 functionalities

printFileDefault: This method prints a specified file to either a default or named printer. It checks if the file exists and if the path is valid, then creates a print job using the appropriate printer. After attempting to print, it performs an action based on the provided flag—keeping the file, moving it to an archive directory, or deleting it.

transferFileToArchive: Moves a file to an "archive" subdirectory within its parent directory. It first verifies that the file exists and then ensures the "archive" directory is present before moving the file to this directory.

deleteFile: Deletes a specified file from the filesystem. It checks if the file exists before attempting to delete it and handles any exceptions that may arise during the deletion process.

ValidateCredentials: Validates user credentials by decrypting an encrypted password using the AES class. This method returns a boolean indicating whether the credentials are valid.

getLogPath: Retrieves the log path from a properties file. It reads the file and returns the value associated with the "log.path" property, handling any IOExceptions that occur.

getAllowedExtensions: Loads a list of allowed file extensions from a properties file. It parses the file to extract and return a list of extensions specified under the "allowedExtensions" property.

ValidateExtension: Checks if a file's extension is among the allowed extensions specified in a properties file. It retrieves the list of allowed extensions and compares it with the extension of the given file name.

getFileExtension: Extracts and returns the file extension (including the dot) from a given file name. It handles cases where the file name may not have an extension or where the dot is the first character.

popNotification: Displays a notification with a specified message. It utilizes the NotificationUtils class to show the notification and logs the event.

Properties File

```
# AES-256 Configuration
aes.key=2b7e151628aed2a6abf7158809cf4f3c76418f4a3f9870e8ad4b80b2d8e1b0f2
aes.IV=000102030405060708090a0b0c0d0e0f
aes.username=ac89fe6a2686530392c5886d4f62efe2
aes.password=8b8a7004c5f0a80f03e7043f01cb0184
log.path=C:/Users/E14/Desktop/LogFile/printer.log
allowedExtensions=.txt .pdf .csv .html .htm .xls .xlsx .doc .docx .ppt .pptx .png .jpg .jpeg .gif
```

Figure 38 Properties File

- **aes.key**: This is the AES-256 encryption key used for encrypting and decrypting data. It should be a 32-byte (64-character hexadecimal) string.
- **aes.IV**: The Initialization Vector (IV) used in AES encryption. It ensures that encryption is unique each time and should be a 16-byte (32-character hexadecimal) string.
- **aes.username**: An encrypted username used for authentication purposes. It is expected to be decrypted for validation.
- **aes.password**: An encrypted password used for authentication purposes. It is also expected to be decrypted for validation.
- **log.path**: The file path where log files are saved. This configuration points to C:/Users/E14/Desktop/LogFile/printer.log.
- **allowedExtensions**: A space-separated list of file extensions that are permitted for processing. This list includes common document and image formats.

Log File

```
Aug 12, 2024 11:14:41 AM commandhandler.CommandHandler receivePath
INFO: command: transfer
Aug 12, 2024 11:14:41 AM commandhandler.CommandHandler receivePath
INFO: printer name:
Aug 12, 2024 11:14:41 AM commandhandler.CommandHandler receivePath
INFO: directory: C:\Users\E14\Desktop\TestingPrintFinal
Aug 12, 2024 11:14:41 AM commandhandler.CommandHandler receivePath
INFO: flag :keep
Aug 12, 2024 11:14:41 AM commandhandler.CommandHandler receivePath
INFO: file to retrieve:
Aug 12, 2024 11:14:41 AM commandhandler.CommandHandler receivePath
INFO: message:
Aug 12, 2024 11:14:41 AM commandhandler.CommandHandler receivePath
INFO: username: demon
Aug 12, 2024 11:14:41 AM commandhandler.CommandHandler receivePath
INFO: encrypted password: 8b8a7004c5f0a80f03e7043f01cb0184
Aug 12, 2024 11:14:41 AM cryptography.AES Decrypt
INFO: user authenticated
Aug 12, 2024 11:14:41 AM commandhandler.CommandHandler receivePath
INFO: directory created: C:\Users\E14\Desktop\TestingPrintFinal
Aug 12, 2024 11:14:41 AM commandhandler.CommandHandler receivePath
INFO: file name: AUTOPRINT.docx
Aug 12, 2024 11:14:41 AM commandhandler.CommandHandler receivePath
INFO: file transfered
Aug 12, 2024 11:14:41 AM commandhandler.CommandHandler receivePath
INFO: file name: AUTOPRINT.xlsx
Aug 12, 2024 11:14:41 AM commandhandler.CommandHandler receivePath
INFO: file transfered
Aug 12, 2024 11:14:41 AM commandhandler.CommandHandler receivePath
INFO: file name: CommandHandlerProject.pptx
Aug 12, 2024 11:14:41 AM commandhandler.CommandHandler receivePath
INFO: file transfered
```

Figure 39 Log File

Log Overview:

1. **Command Execution:**
    - **Timestamp:** Aug 12, 2024 11:14:41 AM
    - **Component:** commandhandler.CommandHandler
    - **Action:** receivePath
    - **Info: command: transfer**
    - **Explanation:** A transfer command is being processed.
2. **Printer Information:**
    - **Info: printer name:**
    - **Explanation:** No specific printer is designated for this operation.

3. **Directory Details:**
    - **Info: directory: C:\Users\E14\Desktop\TestingPrintFinal**
    - **Explanation:** Files are being managed in the specified directory.
4. **Flag Value:**
    - **Info: flag: keep**
    - **Explanation:** The flag keep indicates that files should be retained after processing.
5. **File Retrieval:**
    - **Info: file to retrieve:**
    - **Explanation:** No specific file is mentioned for retrieval.
6. **Additional Message:**
    - **Info: message:**
    - **Explanation:** No additional message is logged.
7. **User Authentication:**
    - **Info: username: demon**
    - **Info: encrypted password: 8b8a7004c5f0a80f03e7043f01cb0184**
    - **Explanation:** The username demon is used with an encrypted password for authentication.
8. **Authentication Status:**
    - **Timestamp:** Aug 12, 2024 11:14:41 AM
    - **Component:** cryptography.AES Decrypt
    - **Info: user authenticated**
    - **Explanation:** The user has been successfully authenticated using AES decryption.
9. **Directory Creation:**
    - **Info: directory created: C:\Users\E14\Desktop\TestingPrintFinal**
    - **Explanation:** The specified directory has been created.
10. **File Transfer Status:**
    - **Info: file name: AUTOPRINT.docx**
    - **Info: file transferred**

- **Explanation:** The file AUTOPRINT.docx has been successfully transferred.

11. **Next File Transfer:**
    - **Info: file name: AUTOPRINT.xlsx**
    - **Info: file transferred**
    - **Explanation:** The file AUTOPRINT.xlsx has been successfully transferred.
12. **Final File Transfer:**
    - **Info: file name: CommandHandlerProject.pptx**
    - **Info: file transferred**
    - **Explanation:** The file CommandHandlerProject.pptx has been successfully transferred.

Dependencies

The project relies on several Java libraries to function, including commons-collections4-4.4, commons-compress-1.26.2, commons-io-2.16.1, commons-logging-1.2, jackson-annotations-2.17, jackson-core-2.17.0, jackson-databind-2.17.0, jackson-dataformat-xml-2.17.0, javax.servlet-3.1, json-20240303, log4j-api-2.20.0, log4j-core-2.20.0, ojdbc8, poi-4.1.2, poi-ooxml-4.1.2, poi-ooxml-schemas-4.1.2, poi-scratchpad-4.1.2, servlet-api, SparseBitSet-1.2, and xmlbeans-3.1.0

Limitations

The project currently has several limitations that should be noted. It only supports a specific set of file extensions, which include .txt, .pdf, .csv, .html, .htm, .xlsx, .docx, .pptx. For document files like .docx, it handles only a single page of text, while for Excel files (.xlsx), it processes only one sheet. Presentations in .pptx format are limited to one slide. Additionally, the system is not designed to handle high loads effectively; performance may degrade under heavy or concurrent usage. These constraints should be considered when planning for system deployment and use, with potential future updates needed to address these limitations and improve scalability.

Conclusion

The project effectively manages file operations, including printing, transferring, and deleting, utilizing AES encryption for secure user authentication. It ensures robust functionality through detailed logging of actions and statuses. The integration with the application server enhances the system by allowing it to handle diverse commands and requests efficiently, ensuring seamless communication between the user interface and the connected printer.

Future improvements should focus on thorough testing to catch edge cases, enhancing file management features to support more file types and printer configurations, and bolstering security measures by refining encryption and key management. Additionally, comprehensive documentation and user training should be provided to facilitate smooth operation and troubleshooting. Expanding support for various application server setups will further improve system flexibility and compatibility. These steps will enhance the system’s reliability, security, and user experience, ensuring it meets evolving needs and standards.