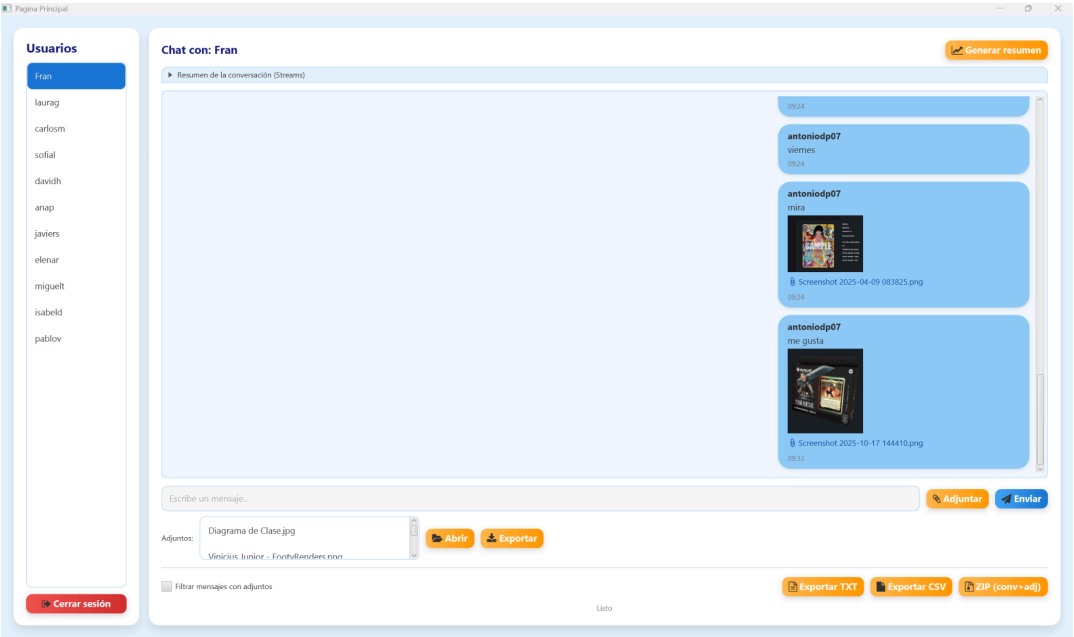
IES Francisco de los Ríos

Alumno	**Antonio Delgado Portero**
Asignatura	**Acceso a datos**
Curso	**2 DAM**
Año	**2025-2026**
Título de la práctica	**Proyecto: Chat Offline**
GIT	**Enlace**

1. Introducción

La presente memoria técnica documenta el desarrollo de la aplicación de escritorio "**Chat Offline XML**", implementada sobre la plataforma **JavaFX**. El objetivo principal del proyecto es proporcionar un sistema de comunicación mediante chat para usuarios registrados en un entorno local, sin dependencia de conectividad a redes externas. La aplicación garantiza la persistencia de los datos de usuario y las conversaciones mediante el uso de archivos **XML** como mecanismo de almacenamiento local.



El sistema va más allá de la mensajería textual básica, incorporando funcionalidades para la **gestión de archivos adjuntos**. Los usuarios pueden asociar ficheros (imágenes, documentos PDF, ofimáticos, etc.) a sus mensajes. Para mantener la integridad y prevenir problemas de almacenamiento, se implementan **validaciones** sobre estos adjuntos, restringiendo su tamaño máximo y tipo de archivo permitido.

Con el fin de facilitar la portabilidad y el análisis externo de las conversaciones, se han integrado diversas **capacidades de exportación**. Los historiales de chat pueden ser exportados a formatos estándar como texto plano (**TXT**) o valores separados por comas (**CSV**). Adicionalmente, se ofrece la opción de empaquetar una conversación completa, incluyendo el texto y todos los archivos adjuntos asociados, en un único archivo comprimido **ZIP**. La aplicación también incluye un apartado de **análisis estadístico**, capaz de generar resúmenes sobre la participación de los usuarios y la frecuencia de términos utilizados en el chat, aprovechando la API de Streams de Java.

La **interfaz gráfica de usuario (GUI)** destaca la implementación del área de visualización de mensajes mediante un control ListView de JavaFX, renderizando cada mensaje dentro de una celda personalizada (ChatMessageCell) que adopta la forma de **burbuja de chat**, un elemento visual muy reconocido en aplicaciones de mensajería modernas.

Este documento detallara las tecnologías empleadas, la arquitectura de software adoptada, el modelo de datos, las funcionalidades implementadas y otros aspectos técnicos relevantes del proyecto.

2. Tecnologías utilizadas

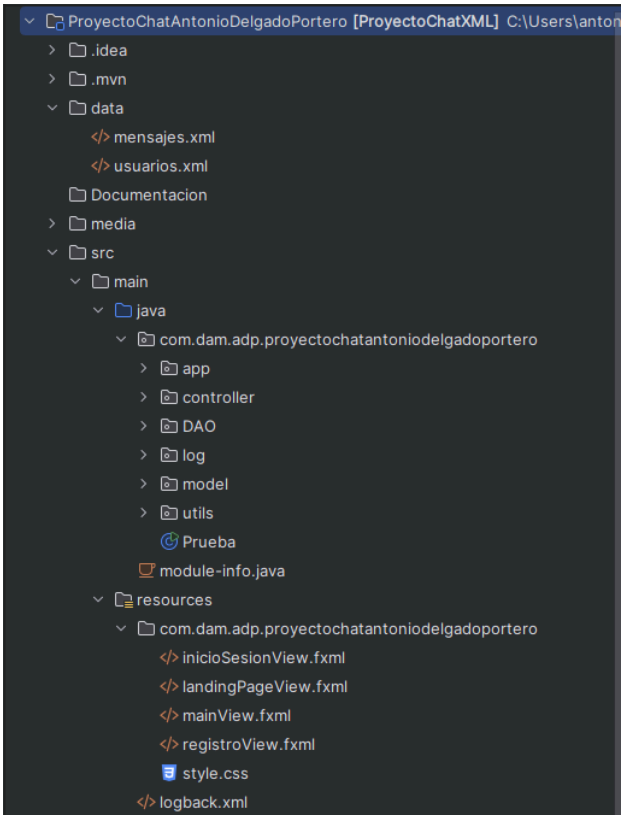
La implementación del proyecto "Chat Offline XML" se ha apoyado en un conjunto de tecnologías consolidadas dentro del ecosistema Java, seleccionadas por su comodidad para el desarrollo de aplicaciones de escritorio robustas y mantenibles:

- **Java (JDK 21+)**: Como lenguaje de programación principal, se ha utilizado una versión reciente del Java Development Kit (específicamente, configurado para el target de compilación 24 en Maven), lo que permite el uso de características modernas del lenguaje y asegura compatibilidad con las bibliotecas empleadas.
- **JavaFX**: Framework principal para la construcción de la Interfaz Gráfica de Usuario (GUI). Proporciona la base para la creación de ventanas, escenas y un amplio catálogo de controles interactivos (**ListView, TextField, Button, CheckBox, TitledPane,** etc.). Se utilizan explícitamente los módulos **javafx-controls y javafx-fxml**.
- **FXML**: Lenguaje declarativo basado en XML, utilizado para definir la estructura y disposición de los elementos de la interfaz gráfica. Permite separar el diseño de la interfaz de la lógica de la aplicación implementada en los controladores Java, facilitando el desarrollo y mantenimiento.
- **CSS (Cascading Style Sheets)**: Empleado para definir la apariencia visual (**styling**) de los componentes JavaFX. Permite personalizar colores, fuentes, márgenes, bordes, efectos (sombras, gradientes), etc., logrando una estética coherente y moderna en toda la aplicación.
- **Maven**: Herramienta de gestión del ciclo de vida del proyecto y de dependencias. Se utiliza para definir las bibliotecas externas requeridas (JAXB, Logback, jBCrypt, ControlsFX, etc.), descargar automáticamente dichas bibliotecas y orquestar las fases de compilación, empaquetado y ejecución.

- **JAXB (Java Architecture for XML Binding)**: API estándar de Java para la serialización y deserialización de objetos Java a/desde XML. Es la tecnología central para la persistencia de datos, permitiendo guardar el estado de los objetos **Usuario y Mensaje** en los archivos **usuarios.xml y mensajes.xml**, y recuperarlos posteriormente.
- **SLF4J (Simple Logging Facade for Java) + Logback**: Framework estándar para el registro de eventos (logging). SLF4J proporciona una abstracción (fachada), mientras que Logback actúa como la implementación concreta que gestiona la salida de los mensajes de log (a consola y al archivo **chat.log**). Es esencial para el diagnóstico de errores y el seguimiento del flujo de ejecución.
- **jBCrypt**: Biblioteca externa utilizada para implementar el hashing seguro de las contraseñas de los usuarios. Utiliza el algoritmo bcrypt, que incorpora *salting* automático y un factor de trabajo configurable para dificultar ataques de fuerza bruta y de diccionario.
- **Java Streams API**: Utilizada en la clase **StreamUtils** para el procesamiento declarativo y funcional de las colecciones de mensajes. Permite realizar operaciones como filtrado, mapeo, agrupación y cálculo de agregados (conteos, frecuencias) de manera eficiente y legible, especialmente para generar las estadísticas del chat.
- **ControlsFX**: Biblioteca que enriquece JavaFX con controles y funcionalidades adicionales. En este proyecto, se utiliza específicamente el componente **Glyph** para integrar fácilmente iconos de la fuente FontAwesome en los botones de la interfaz, mejorando la usabilidad y la estética.

3. Arquitectura del sistema

La estructura de la aplicación se ha diseñado siguiendo principios de **modularidad** y **separación de conceptos**, facilitando su comprensión, mantenimiento y posible extensión futura. El paradigma arquitectónico principal adoptado es el patrón **Modelo-Vista-Controlador (MVC)**, adaptado a las convenciones de JavaFX.



- **Modelo (Model)**: Comprende las clases que representan las entidades de datos (**Usuario, Mensaje**) y las estructuras que las agrupan (**ListaUsuarios, Mensajes**), así como clases que gestionan el estado global (**Sesion**). Encapsula los datos y la lógica de negocio fundamental, independientemente de cómo se presenten o almacenen.
- **Vista (View)**: Constituye la interfaz gráfica con la que interactúa el usuario final. Se define mediante archivos **FXML**, que describen la estructura jerárquica de los componentes visuales y su disposición. La apariencia se personaliza mediante **CSS**.
- **Controlador (Controller)**: Actúa como el "pegamento" entre la vista y el Modelo. Cada archivo FXML está asociado a una clase controladora Java que contiene la lógica para manejar los eventos de la interfaz (clics de botón, selecciones de lista, etc.). El controlador interactúa con el Modelo (a través de los DAO) para recuperar o modificar datos y actualiza la Vista para reflejar estos cambios.

Adicionalmente, se emplean otros patrones y estructuras para mejorar la organización:

- **DAO (Data Access Object)**: Se utiliza este patrón para abstraer y encapsular el acceso a la fuente de datos (archivos XML). Las clases **UsuarioDAO y MensajeDAO** contienen toda la

lógica relacionada con JAXB y la manipulación de los archivos **usuarios.xml y mensajes.xml**. Esto desacopla a los controladores de los detalles de implementación de la persistencia.

- **Clases de Utilidad (utils)**: Se agrupan funcionalidades transversales y reutilizables en clases estáticas dentro del paquete **utils**. Esto incluye manejo de archivos (**FileManager**), serialización XML (**XMLManager**), hashing de contraseñas (**PasswordManager**), procesamiento con Streams (**StreamUtils**), validaciones genéricas y navegación (**Utils**), conversión de tipos para JAXB (**LocalDateTimeAdapter**) y la celda personalizada del chat (**ChatMessageCell**).
- **Punto de Entrada (app)**: El paquete app contiene las clases **Launcher** y **Aplicacion**, responsables únicamente de iniciar la aplicación JavaFX y cargar la primera vista FXML.

3.1. Detalle: **FileManager** y su Interacción con **MainController**

La clase de utilidad **FileManager** desempeña un rol fundamental al **centralizar y abstraer todas las interacciones con el sistema de archivos**. Esta separación es clave para mantener la cohesión del **MainController**, permitiéndole concentrarse en la lógica de presentación y flujo de la aplicación.

Responsabilidades Específicas de FileManager:

- **Validación de Archivos (validarArchivo)**: Implementa la lógica para verificar si un **File** cumple con los criterios definidos: existencia, tamaño máximo (definido por la constante **TAMAÑO_MAXIMO_BYTES**) y si su extensión pertenece a la lista de tipos permitidos (definida en **EXTENSIONES_PERMITIDAS**).
- **Generación de Nombres Únicos (generarNombreUnico)**: Crea un nombre de archivo único añadiendo un UUID al nombre original, preservando la extensión. Esto es esencial para evitar colisiones al guardar múltiples archivos con el mismo nombre en la carpeta **media/**.
- **Operaciones de Transferencia de Archivos (guardarArchivo, exportarArchivo)**: Contiene la lógica para copiar archivos utilizando **InputStream** y **OutputStream** (específicamente **BufferedInputStream** y **BufferedOutputStream** para eficiencia). **guardarArchivo** copia un archivo de origen a la carpeta **media/** con un nombre específico, mientras que **exportarArchivo** copia un archivo (generalmente desde **media/)** a una carpeta seleccionada por el usuario.
- **Generación de Archivos de Texto (exportarAArchivoTexto, exportarAArchivoCsv, exportarEstadisticas)**: Proporciona métodos para escribir listas de **Mensaje** o **String** en archivos **.txt** o **.csv**, encargándose del formato adecuado (timestamps, separadores definidos en constantes como **CSV_SEPARADOR**).

- **Creación de Archivos ZIP (crearArchivoZip)**: Implementa la lógica para generar un archivo **.zip** utilizando **ZipOutputStream**. Añade una entrada de texto (**ZipEntry** con nombre **NOMBRE_TEXTO_CONVERSACION**) con el contenido de la conversación y luego itera sobre los mensajes para añadir cada archivo adjunto físico (localizado mediante **getRutaMedia**), leyéndolo por bloques (buffer) para optimizar el uso de memoria.
- **Interacción con el Escritorio (abrirArchivo)**: Utiliza **java.awt.Desktop.getDesktop().open()** para solicitar al sistema operativo que abra un archivo con su aplicación predeterminada.
- **Utilidades de Ruta y Tipo (getRutaMedia, detectarMimeType)**: Proporciona métodos para construir rutas dentro de la carpeta **media/** de forma independiente del sistema operativo (**File.separator**) y para intentar detectar el tipo MIME de un archivo basado en sus bytes iniciales (**URLConnection.guessContentTypeFromStream**).

Flujo de Interacción MainController -> FileManager:

MainController delega las operaciones de archivo a **FileManager** de la siguiente manera:

Selección de Adjunto: **MainController** usa **FileChooser** y luego invoca **FileManager.validarArchivo**.

Guardado de Adjunto: Lo realiza **MensajeDAO**, que a su vez llama a **FileManager.generarNombreUnico** y **FileManager.guardarArchivo**.

Abrir/Exportar Adjunto Seleccionado: **MainController** usa su método **localizarArchivoAdjunto** (que internamente puede usar **FileManager.getRutaMedia**) para obtener el File y luego llama a **FileManager.abrirArchivo** o **FileManager.exportarArchivo**.

Exportaciones (TXT, CSV, ZIP, Estadísticas): **MainController** prepara los datos (mensajes o estadísticas), usa **mostrarDialogoExportar** para obtener el **File** destino del usuario, y finalmente invoca el método específico de **FileManager** (**exportarAArchivoTexto, exportarAArchivoCsv, crearArchivoZip, exportarEstadisticas**) para realizar la escritura.

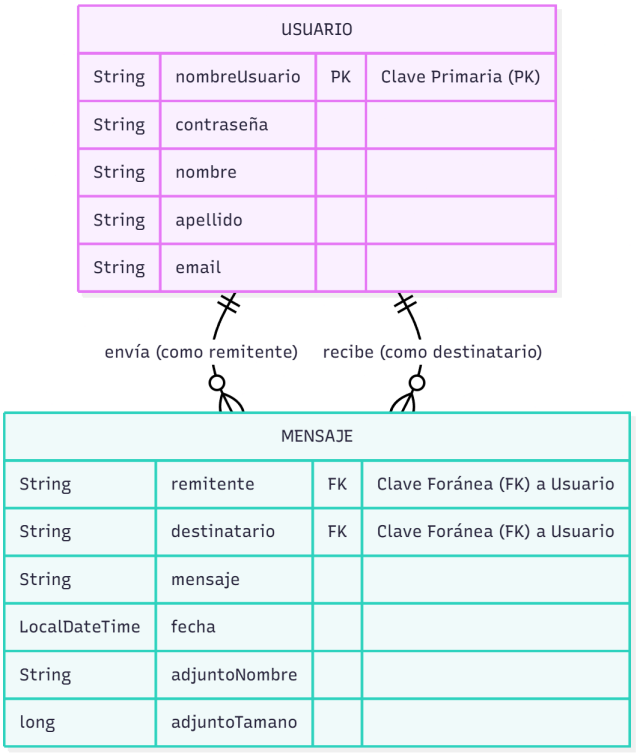
Esta arquitectura asegura que la lógica compleja de manejo de archivos esté encapsulada y sea reutilizable, manteniendo el controlador más limpio.

4. Modelo de Datos

El modelo de datos define la estructura de la información que maneja la aplicación. Las entidades principales son:

- **Usuario**: Representa a un usuario registrado.
    - **Atributos**:
        - **nombreUsuario** (String): Identificador único y clave primaria conceptual.
        - **contraseña** (String): Almacena el hash **bcrypt** de la contraseña.
        - **nombre** (String): Nombre de pila.
        - **apellido** (String): Apellido.
        - **email** (String): Dirección de correo electrónico.
    - **Persistencia**: Serializado como elemento **<usuario>** en **data/usuarios.xm**l. Incluye Javadoc explicativo.
- **Mensaje**: Representa una unidad de comunicación.
    - **Atributos**:
        - **remitente** (String): Clave foránea referenciando **Usuario**.**nombreUsuario**.
        - **destinatario** (String): Clave foránea referenciando **Usuario.nombreUsuario**.
        - **mensaje** (String): Contenido textual.
        - **fecha** (LocalDateTime): Timestamp de envío, adaptado para XML con **LocalDateTimeAdapter**.
        - **adjuntoNombre** (String, opcional): Nombre original del archivo adjunto.
        - **adjuntoRuta** (String, opcional): Ruta relativa (bajo **media/**) donde se almacena la copia del adjunto.
        - **adjuntoTamano** (long, opcional): Tamaño en bytes del adjunto.
        - **adjuntoTipo** (String, opcional): Extensión o tipo MIME del adjunto.
    - **Persistencia**: Serializado como elemento **<mensaje>** en **data/mensajes.xml**. Incluye Javadoc explicativo.
- **ListaUsuarios** y **Mensajes**: Clases contenedoras anotadas con **@XmlRootElement** y **@XmlElement** para facilitar a JAXB el manejo de las listas de **Usuario** y **Mensaje** al serializar/deserializar los archivos XML completos.
- **Sesion**: Implementa el patrón **Singleton** para garantizar una única instancia que gestiona el estado de la sesión, manteniendo una referencia al **Usuario** actualmente autenticado (**usuairoLogueado**).

Diagrama Entidad-Relación (ER):



El modelo refleja una relación **uno a muchos (1:N)** bidireccional entre **Usuario** y **Mensaje**: un usuario puede enviar y recibir múltiples mensajes, pero cada mensaje pertenece a un único remitente y un único destinatario. Las claves foráneas **remitente** y **destinatario** en **Mensaje** establecen esta vinculación con la clave primaria **nombreUsuario** de **Usuario**.

5. Casos de uso

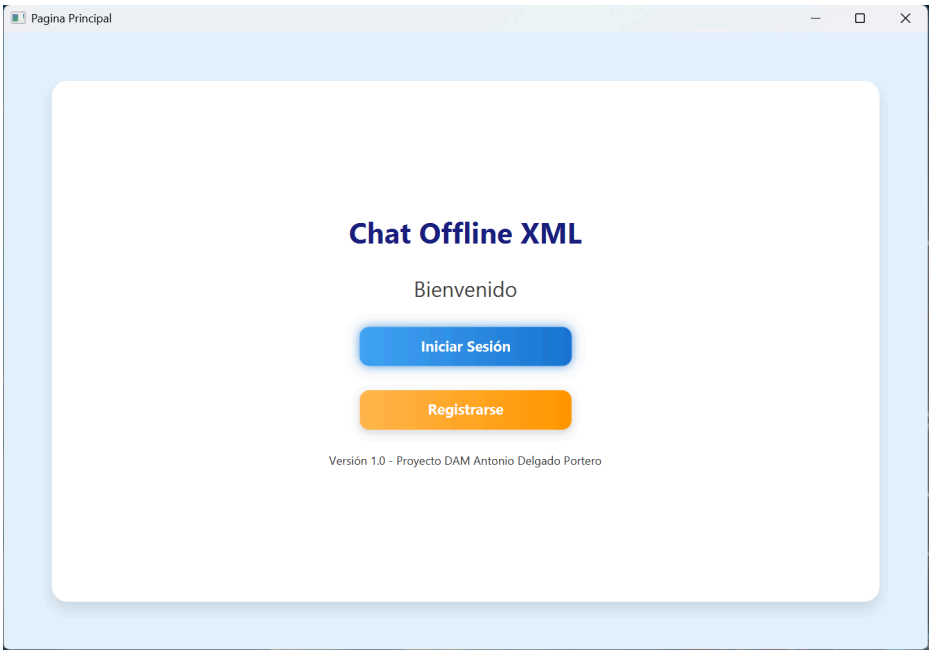
La aplicación soporta las siguientes funcionalidades principales desde la perspectiva del usuario:

- **CU01: Registrar Usuario**: Permite a un nuevo usuario crear una cuenta.
- **CU02: Iniciar Sesión**: Permite a un usuario existente autenticarse.
- **CU03: Visualizar Contactos**: Muestra la lista de usuarios registrados (excluyendo el propio).
- **CU04: Seleccionar Contacto**: Permite elegir un usuario de la lista para interactuar.
- **CU05: Visualizar Conversación**: Carga y muestra el historial de mensajes con el contacto seleccionado.
- **CU06: Enviar Mensaje Textual**: Compone y envía un mensaje de texto.

- **CU07: Adjuntar Archivo**: Selecciona un archivo local para asociarlo a un mensaje.
- **CU08: Enviar Mensaje con Adjunto**: Envía un mensaje que incluye el archivo adjunto seleccionado.
- **CU09: Filtrar Conversación por Adjuntos**: Permite mostrar únicamente los mensajes que contienen adjuntos.
- **CU10: Abrir Archivo Adjunto**: Intenta abrir un adjunto de la conversación con la aplicación predeterminada del sistema.
- **CU11: Exportar Archivo Adjunto**: Guarda una copia local de un adjunto específico.
- **CU12: Exportar Conversación (TXT)**: Guarda el historial del chat en formato .txt.
- **CU13: Exportar Conversación (CSV)**: Guarda el historial del chat en formato .csv.
- **CU14: Exportar Conversación Completa (ZIP)**: Genera un archivo .zip con el texto del chat y todos sus adjuntos.
- **CU15: Generar/Exportar Estadísticas**: Calcula métricas sobre la conversación y permite guardarlas en .txt.
- **CU16: Cerrar Sesión**: Finaliza la sesión activa del usuario.

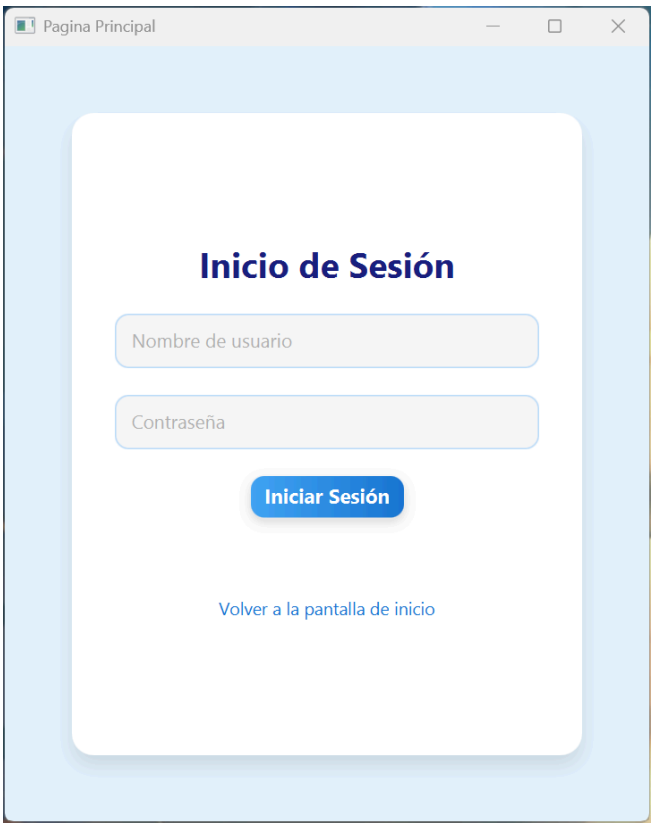
6. Interfaces de usuario (GUI)

La Interfaz Gráfica de Usuario (GUI) se ha desarrollado utilizando **JavaFX**, con la estructura definida en archivos **FXML** y la apariencia personalizada mediante **CSS**. Consta de las siguientes vistas principales:

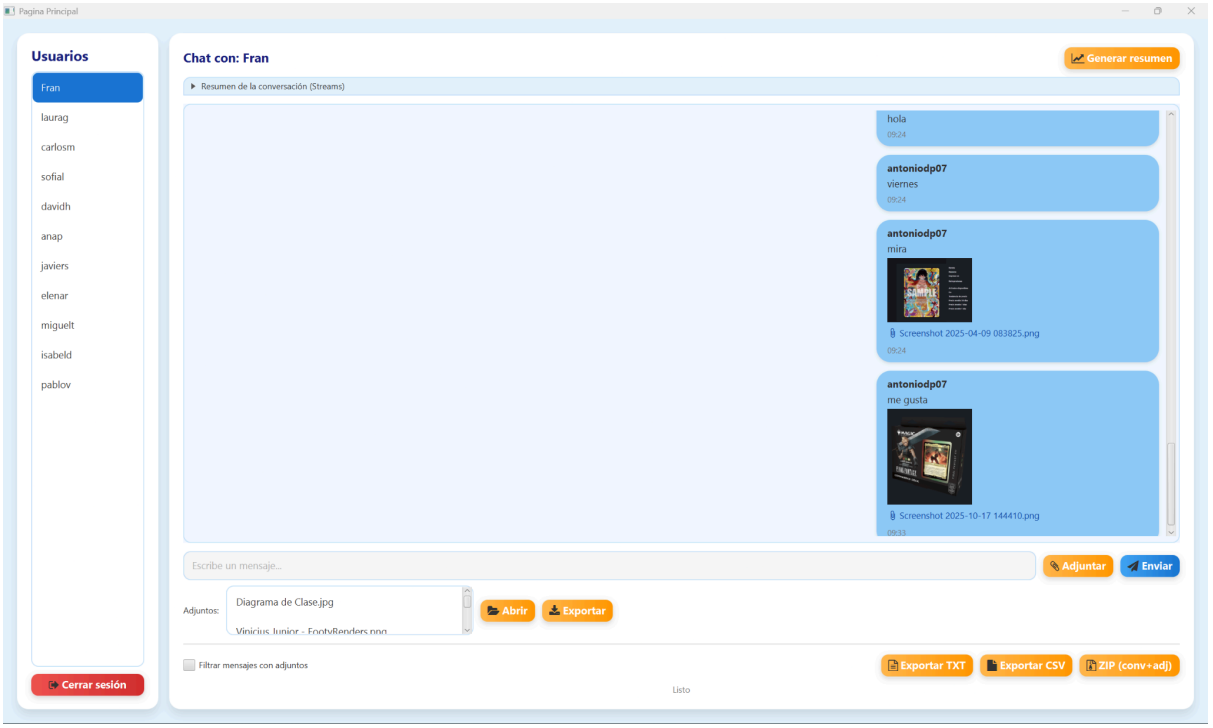


1. **Vista de Bienvenida (landingPageView.fxml)**: Pantalla inicial que recibe al usuario, presentando el título de la aplicación y dos opciones principales mediante botones: "Iniciar Sesión" y "Registrarse".

2. **Vista de Registro (registroView.fxml)**: Contiene el formulario para la creación de nuevas cuentas. Incluye campos **TextField** para **nombreUsuario**, **nombre**, **apellido**, **email** y un **PasswordField** para **contraseña**. Dispone de un botón "Registrar" para enviar los datos y un Hyperlink para regresar a la pantalla de bienvenida. Una **Label** (**lblMensaje**) proporciona feedback sobre el proceso.



3. **Vista de Inicio de Sesión (inicioSesionView.fxml)**: Presenta un formulario simplificado para la autenticación, solicitando únicamente **nombreUsuario** (**TextField**) y **contraseña** (**PasswordField**). Incluye el botón "Iniciar Sesión" y el **Hyperlink** para volver. También utiliza **lblMensaje** para feedback.



4. **Vista Principal del Chat (mainView.fxml)**: Es la interfaz central de la aplicación, estructurada en dos paneles principales dentro de un **HBox**:
    - **Panel Lateral (VBox menuLateral)**: Situado a la izquierda, contiene un **Label** "Usuarios" y un **ListView** (**listaUsuarios**) que muestra los contactos disponibles. En la parte inferior, se encuentra el botón "Cerrar sesión", ahora con un icono **SIGN_OUT** de FontAwesome.
    - **Panel de Chat (VBox panelChat)**: Ocupa el área principal y contiene, de arriba abajo:
        - Una cabecera (**HBox**) con **lblUsuarioChat** (indicando el interlocutor) y el botón "Generar resumen" (con icono **LINE_CHART**).
        - Un **TitledPane** colapsable para mostrar las estadísticas de la conversación en un **GridPane**.
        - El área principal de visualización de mensajes: un **ListView** (**lvChat**) con **styleClass="message-list"**, que utiliza **ChatMessageCell** para renderizar los mensajes como burbujas.
        - La zona de entrada de mensajes (**VBox**), compuesta por:
            - Un **HBox** con el **TextField** (**txtMensaje**) para escribir, el botón "Adjuntar" (icono **PAPERCLIP**) y el botón "Enviar" (icono **SEND**).
            - Un HBox para gestionar los adjuntos *de la conversación actual*, mostrando un **Label**, un **ListView** (**listaAdjuntos**) con los nombres de

los adjuntos, y los botones "Abrir" (icono **FOLDER_OPEN**) y "Exportar" (icono **DOWNLOAD**).

- Un **Separator**.
- Una barra inferior (**HBox**) con el **CheckBox** (**chkSoloAdjuntos**) para filtrar, y los botones de exportación: "Exportar TXT" (icono **FILE_TEXT_ALT**), "Exportar CSV" (icono **FILE**), y "ZIP (conv+adj)" (icono **FILE_ARCHIVE_ALT**).
- Una **Label** (**lblEstado**) en la base para mostrar mensajes de estado.

El archivo **style.css** define la apariencia visual completa, aplicando un tema minimalista y moderno, e incluyendo ahora las reglas necesarias para estilizar las burbujas de chat (**.message-list, .message-bubble, .own-message, .other-message, .timestamp**).

7. Controladores

Cada vista FXML está gestionada por una clase controladora Java que implementa la lógica de interacción y la comunicación con las capas inferiores (DAO, Modelo, Utils):

- **LandingPageController**: Gestiona los eventos de los botones "Iniciar Sesión" y "Registrarse", invocando **Utils.cambiarEscena** para efectuar la navegación a las vistas correspondientes.
- **RegistroController**: Maneja el formulario de registro. El método **registrarUsuario** recopila los datos, realiza validaciones (campos vacíos, formato de email usando **Utils.validarEmail**), hashea la contraseña (**PasswordManager.hashPassword**), intenta persistir el nuevo usuario mediante **UsuarioDAO.añadirUsuario**, y actualiza **lblMensaje** para informar al usuario del resultado.
- **InicioSesionController**: Gestiona la autenticación. El método **iniciarSesion** obtiene las credenciales, las verifica llamando a **UsuarioDAO.validarCredenciales**, y en caso de éxito, establece la sesión global (**Sesion.getInstancia().iniciarSesion**) y navega a la vista principal (**mainView.fxml**) usando **Utils.cambiarEscena**. Proporciona feedback en **lblMensaje** en caso de fallo.
- **MainController**: Es el controlador principal y más complejo, responsable de la funcionalidad del chat:
    - **initialize**: Realiza la configuración inicial de la vista: obtiene el usuario logueado de **Sesion**, carga la lista de contactos desde **UsuarioDAO**, establece un listener en **listaUsuarios** para detectar la selección de un interlocutor, y configura el **cellFactory** del **ListView** de chat (**lvChat**) para utilizar la celda personalizada **ChatMessageCell**.

También asocia las acciones a los diferentes botones (adjuntar, abrir/exportar adjuntos, exportar chat, etc.) y configura el envío con la tecla Enter.

- **mostrarMensajes**: Método clave para visualizar la conversación. Obtiene los mensajes entre el usuario logueado y el seleccionado (**MensajeDAO.listarMensajesEntre**), aplica el filtro **chkSoloAdjuntos** si está activo, prepara una **ObservableList<Mensaje>** con los mensajes a mostrar, la asigna a **lvChat.setItems()**, realiza el scroll automático al último mensaje y actualiza la **listaAdjuntos** lateral con los nombres de los adjuntos encontrados.
- **enviarMensaje**: Gestiona el envío. Obtiene el texto del **txtMensaje**, comprueba si hay un **adjuntoSeleccionado**, invoca el método apropiado de **MensajeDAO** (**enviarMensaje** o **enviarMensajeConAdjunto**), y finalmente llama a **mostrarMensajes** y **cargarEstadisticas** para refrescar la interfaz. Limpia los campos de entrada tras el envío.
- **Gestión de Estadísticas (cargarEstadisticas, generarResumen, generaEstadisticasTexto)**: Calcula las métricas (total, por usuario, palabras top) utilizando **StreamUtils**, las muestra en las **Label** correspondientes y las formatea para su exportación a través de **FileManager**.
- **Gestión de Adjuntos (seleccionarAdjunto, localizarArchivoAdjunto, abrirAdjunto, exportarAdjunto)**: Orquesta el ciclo de vida de los adjuntos: selección (usando **FileChooser**), validación (delegando a **FileManager**.**validarArchivo**), localización del archivo físico (con el método extraído **localizarArchivoAdjunto**) y ejecución de acciones (abrir/exportar, delegando a FileManager).
- **Gestión de Exportaciones (exportarTxt, exportarCsv, exportarZip, mostrarDialogoExportar)**: Maneja los eventos de los botones de exportación. Utiliza el método refactorizado mostrarDialogoExportar para obtener la ruta de destino del usuario y luego invoca el método correspondiente de **FileManager** para generar el archivo en el formato solicitado.

8. DAO (Data Access Object)

La capa de Acceso a Datos (DAO) proporciona una interfaz abstracta para interactuar con el mecanismo de persistencia (archivos XML), ocultando los detalles de implementación (JAXB, manejo de archivos) al resto de la aplicación.

- **UsuarioDAO**: Encapsula todas las operaciones relacionadas con los datos de usuario almacenados en data/usuarios.xml.
    - **leerUsuarios()**: Utiliza **XMLManager**.**readXML** para deserializar el archivo XML completo en un objeto ListaUsuarios. Maneja el caso de que el archivo no exista o esté vacío devolviendo una lista vacía.
    - **guardarUsuarios()**: Utiliza **XMLManager**.**writeXML** para serializar el objeto ListaUsuarios (con todos los usuarios) de vuelta al archivo XML.
    - **añadirUsuario()**: Orquesta el proceso de registro: primero llama a **leerUsuarios**, luego comprueba si ya existe un usuario con el mismo **nombreUsuario** o **email**. Si no existe, añade el nuevo **Usuario** a la lista en memoria y llama a **guardarUsuarios** para persistir el cambio. Devuelve **true** si se añadió, **false** si ya existía..
    - **cargarUsuario()**: Busca un usuario específico por su **nombreUsuario** dentro de la lista obtenida de **leerUsuarios**.
    - **validarCredenciales()**: Combina **cargarUsuario** con **PasswordManager.checkPassword** para verificar si la contraseña proporcionada coincide con el hash almacenado para ese usuario.
- **MensajeDAO**: Encapsula las operaciones sobre los mensajes almacenados en **data/mensajes.xml.**
    - **cargarMensajes()**: Utiliza **XMLManager**.**readXML** para deserializar **mensajes**.**xml** en un objeto **Mensajes**.
    - **guardarMensajes()**: Utiliza **XMLManager**.**writeXML** para serializar el objeto **Mensajes** completo.
    - **enviarMensaje()**: Crea un nuevo objeto **Mensaje** con los datos proporcionados y la fecha actual, lo añade a la lista obtenida de **cargarMensajes**, y luego llama a **guardarMensajes**.
    - **enviarMensajeConAdjunto()**: Similar a **enviarMensaje**, pero adicionalmente gestiona el archivo adjunto: llama a **FileManager**.**validarArchivo** (usando las constantes **TAMAÑO_MAXIMO_BYTES** y **EXTENSIONES_PERMITIDAS**), **FileManager**.**generarNombreUnico**, **FileManager**.**guardarArchivo** (para copiar el archivo a **media/**) y **FileManager**.**detectarMimeType** (o **obtenerExtension** como fallback) para poblar los atributos **adjunto*** del objeto **Mensaje** antes de añadirlo y guardar la lista.

- **listarMensajesEntre()**: Carga todos los mensajes (**cargarMensajes**) y luego filtra la lista en memoria para devolver un nuevo objeto **Mensajes** que contiene únicamente las comunicaciones intercambiadas entre los dos **nombreUsuario** especificados.

9. Validaciones y control de errores

La aplicación implementa diversas medidas para garantizar la entrada de datos correcta y gestionar posibles fallos durante la ejecución:

- **Validación de Datos de Entrada**:
  - **Formularios**: Los controladores **RegistroController** e **InicioSesionController** verifican que los campos obligatorios no estén vacíos antes de procesar los datos.
  - **Formato Email**: Se utiliza una expresión regular en **Utils**.**validarEmail** para una comprobación básica del formato del correo electrónico durante el registro.
  - **Unicidad de Usuario**: **UsuarioDAO**.**añadirUsuario** previene el registro de usuarios con **nombreUsuario** o **email** ya existentes.
- **Validación de Archivos Adjuntos**: Antes de procesar un archivo adjunto (al seleccionar o al enviar), se invoca **FileManager.validarArchivo** para comprobar que el archivo existe, no excede el **TAMAÑO_MAXIMO_BYTES** y tiene una extensión incluida en **EXTENSIONES_PERMITIDAS**.
- **Manejo de Excepciones**:
  - **Operaciones de Fichero (I/O)**: La mayoría de los métodos en **FileManager** que realizan operaciones de lectura/escritura (copiar, exportar, ZIP) capturan **IOException** y devuelven un boolean indicando éxito/fracaso, o lanzan **RuntimeException** en casos críticos. Los errores se registran mediante **log.error()**.
  - **Operaciones XML (JAXB)**: **XMLManager** gestiona **JAXBException** (ej., archivo mal formado) y **IOException**. Si **readXML** encuentra un archivo vacío o corrupto, devuelve el objeto por defecto y registra un **log.warn()**. La escritura (**writeXML**) propaga **RuntimeException** si falla.
  - **Navegación FXML**: **Utils.cambiarEscena** incluye un **try-catch** genérico para capturar **IOException** durante la carga de FXML o cualquier otro error inesperado, registrándolo.
- **Retroalimentación al Usuario (Feedback)**: Los controladores utilizan controles **Label** (**lblMensaje** en login/registro, **lblEstado** en la vista principal) para mostrar mensajes claros al

usuario sobre el resultado de sus acciones (éxito, error, cancelación, validación fallida). Se utilizan estilos CSS (ej., texto rojo) para resaltar los errores.

- **Registro (Logging)**: Se utiliza SLF4J/Logback de forma extensiva en controladores, DAOs y utilidades para registrar información relevante (**log.info**), advertencias (**log.warn**) y errores (log.error), incluyendo las trazas de excepciones. Esto es fundamental para la depuración y el análisis post-mortem de problemas.

10. Instalación y despliegue

La aplicación se distribuye como un proyecto Maven estándar. Para compilarla y ejecutarla, se requieren los siguientes pasos y requisitos:

- **Requisitos Previos**:
  - **JDK (Java Development Kit)**: Versión 21 o superior.
  - **Apache Maven**: Versión 3.8 o superior, instalado y accesible desde la línea de comandos.
- **Obtención del Código Fuente**: Clonar o descargar el repositorio del proyecto.
- **Ejecución**: Importar el proyecto como un proyecto Maven en un Entorno de Desarrollo Integrado (IDE) como IntelliJ IDEA, Eclipse o NetBeans. Localizar la clase **com.dam.adp.proyectochatantoniodelgadoportero.app.Launcher** y ejecutar su método **main**. El IDE generalmente se encarga de compilar y configurar el classpath necesario.
- **Estructura de Directorios Requerida**: La aplicación, al ejecutarse, necesita poder leer y escribir en dos subdirectorios relativos al directorio de ejecución:
  - **data/**: Donde se almacenarán los archivos **usuarios.xml** y **mensajes.xml**. Si no existen, las clases **XMLManager** y **FileManager** intentarán crearlos.
  - **media/**: Donde se guardarán las copias de los archivos adjuntos enviados. **FileManager.asegurarDirectorios()** se encarga de crear esta carpeta si no existe. Es importante asegurarse de que la aplicación tenga los permisos de escritura necesarios en su directorio de ejecución.

11. Lista de comprobación de requisitos técnicos

A continuación, se verifica la implementación de las funcionalidades y requisitos técnicos clave del proyecto:

- **Gestión de Usuarios**: Implementado registro, autenticación (login) y persistencia en **usuarios.xml.**

- **Seguridad de Contraseñas**: Utilización de hashing bcrypt mediante jBCrypt.
- **Gestión de Sesión**: Implementado mecanismo Singleton (**Sesion**) para usuario logueado.
- **Interfaz Gráfica (GUI)**: Desarrollada con JavaFX y FXML.
- **Estilo Visual**: Aplicación de CSS para una apariencia personalizada y moderna.
- **Comunicación**: Funcionalidad de chat uno a uno implementada.
- **Visualización de Chat**: Interfaz mejorada con **ListView** y celdas personalizadas (**ChatMessageCell**) para mostrar mensajes como burbujas.
- **Persistencia de Mensajes**: Almacenamiento de conversaciones en **mensajes.xml** mediante JAXB.
- **Funcionalidad de Adjuntos**: Implementada capacidad de adjuntar archivos a los mensajes.
- **Validación de Adjuntos**: Verificación de tamaño y tipo de archivo mediante constantes definidas.
- **Almacenamiento de Adjuntos**: Copia de adjuntos a la carpeta **media/** con nombres únicos (UUID).
- **Visualización de Adjuntos**: Miniaturas para imágenes y enlaces clicables en las burbujas de chat.
- **Apertura de Adjuntos**: Integración con **Desktop.open** para abrir archivos con aplicación predeterminada.
- **Exportación Individual de Adjuntos**: Funcionalidad implementada.
- **Exportación de Conversación (TXT)**: Implementada.
- **Exportación de Conversación (CSV)**: Implementada.
- **Exportación Completa (ZIP)**: Implementada, incluyendo texto y adjuntos.
- **Análisis Estadístico**: Generación de estadísticas (total mensajes, por usuario, palabras comunes) mediante Streams API.
- **Exportación de Estadísticas (TXT)**: Implementada.
- **Filtrado de Mensajes**: Funcionalidad para mostrar solo mensajes con adjuntos.
- **Navegación**: Implementado cambio de escenas entre vistas.
- **Logging**: Sistema de registro de eventos (SLF4J/Logback) configurado y utilizado.
- **Gestión de Proyecto**: Uso de Maven para dependencias y ciclo de vida.
- **Mejoras de Usabilidad**: Iconos en botones (ControlsFX Glyph) y envío de mensajes con tecla Enter.
- **Calidad de Código**: Aplicadas refactorizaciones para reducir duplicación (métodos auxiliares) y mejorar mantenibilidad (constantes).

12. Patrones de diseño y componentes adicionales

El diseño del proyecto se ha basado en varios patrones de diseño y componentes de software reconocidos para mejorar su estructura y calidad:

- **MVC (Modelo-Vista-Controlador)**: Patrón arquitectónico principal que separa los datos (Modelo), la presentación (Vista) y la lógica de control (Controlador), promoviendo la modularidad y facilitando las modificaciones.
- **DAO (Data Access Object)**: Utilizado para abstraer el acceso a la persistencia de datos (XML). Las clases **UsuarioDAO** y **MensajeDAO** centralizan esta lógica.
- **Singleton**: Implementado en la clase **Sesión** para asegurar que exista una única instancia global que gestione la información del usuario autenticado.
- **Utility Class**: Se han creado varias clases (**FileManager, XMLManager, PasswordManager, StreamUtils, Utils**) que agrupan métodos estáticos relacionados, promoviendo la reutilización y evitando la dispersión de lógica común. Se declaran constructores privados para evitar su instanciación.
- **Observer (Implícito en JavaFX)**: El framework JavaFX utiliza internamente este patrón. Un ejemplo en el código es el listener añadido a la propiedad **selectedItemProperty** del **ListView** de usuarios (**listaUsuarios.getSelectionModel()...**), que permite al **MainController** reaccionar automáticamente cuando el usuario selecciona un contacto diferente.
- **Adapter**: La clase **LocalDateTimeAdapter** actúa como un adaptador para JAXB, permitiendo la conversión entre el tipo **java.time.LocalDateTime** (usado en el modelo **Mensaje**) y su representación **String** en el archivo XML.
- **CellFactory**: Se utiliza explícitamente en **MainController (lvChat.setCellFactory(...))** para proporcionar una factoría que crea instancias de **ChatMessageCell**. Este patrón permite desacoplar la lógica de visualización de un elemento individual (**ChatMessageCell**) del propio control **ListView**.
- **API de Streams (Java 8+)**: Empleada en **StreamUtils** para procesar colecciones de mensajes de forma funcional y declarativa, simplificando el código para cálculos estadísticos.
- **Componentes Externos**:
    - **ControlsFX (Glyph)**: Utilizado para añadir iconos a los botones, mejorando la interfaz visual.

13. Futuras mejoras

El estado actual del proyecto constituye una base sólida, pero existen diversas vías para su evolución y mejora:

- **Rendimiento y Escalabilidad (Prioridad Alta)**:
    - **Migración de Persistencia**: Sustituir el almacenamiento basado en archivos XML (JAXB) por una **base de datos relacional embebida** (ej., **SQLite**, H2). Esto eliminaría el cuello de botella de leer/escribir archivos completos para cada operación, permitiendo consultas indexadas y actualizaciones incrementales, lo cual es crucial para manejar un historial de mensajes extenso.
    - **Concurrencia en UI**: Implementar el uso de **javafx.concurrent.Task o Service** para ejecutar todas las **operaciones bloqueantes** (lectura/escritura XML, creación de ZIPs, potencialmente cálculos estadísticos complejos) en **hilos de fondo**. Esto evitaría que la interfaz de usuario se congele durante estas operaciones, mejorando significativamente la responsividad percibida por el usuario. Se deberían añadir indicadores visuales (ej., **ProgressIndicator**) durante la ejecución de estas tareas.
- **Funcionalidad del Chat**:
    - **Chat Grupal**: Extender el modelo y la lógica para permitir conversaciones entre más de dos participantes.
    - **Notificaciones**: Integrar un sistema de notificaciones (ej., usando **Notifications** de ControlsFX) para alertar al usuario de nuevos mensajes cuando la ventana no está en primer plano.
    - **Gestión de Mensajes**: Añadir funcionalidades para **editar** o **eliminar** mensajes ya enviados (requiere lógica adicional en el modelo y DAO, posiblemente con indicadores visuales).
    - **Búsqueda**: Implementar una función para buscar texto dentro del historial de la conversación activa.
    - **Indicador de Escritura**: Añadir una señal visual que indique cuándo el otro usuario está escribiendo un mensaje (requeriría una versión online o un mecanismo de sondeo local).
- **Mejoras en la Interfaz de Usuario (UI/UX)**:
    - **Avatares**: Mostrar imágenes de perfil para los usuarios en la lista de contactos y, opcionalmente, junto a las burbujas de mensaje.

    - **Formato de Texto**: Permite aplicar formatos básicos como **negrita** o *cursiva* a los mensajes.
    - **Previsualización de Adjuntos**: Mejorar la interacción con adjuntos, ofreciendo previsualizaciones más grandes para imágenes o iconos más específicos para otros tipos de archivo. Implementar barras de progreso para la carga/descarga si se migra a un modelo online.
- **Calidad y Mantenimiento**:
    - **Pruebas Automatizadas**: Desarrollar un conjunto robusto de **pruebas unitarias** (JUnit) para las clases de lógica (Utils, DAO, Modelo) y **pruebas de integración** para verificar la interacción entre componentes.
    - **Internacionalización (i18n)**: Refactorizar el código para extraer los textos de la interfaz a archivos de recursos (**.properties**), permitiendo traducir la aplicación a otros idiomas.

14. Backlog del proyecto

ID	Épica / Historia de Usuario (HU)	Descripción / Tareas Principales	Prioridad	Sprint / Fechas
E1	**Gestión de usuarios**	Permitir el registro y almacenamiento de usuarios en XML.	Alta	Sprint 1 (12/Oct)
HU-1	Como usuario, quiero poder registrarme para usar el chat.	- Crear clase Usuario. - Crear usuarios.xml con JAXB. - Implementar añadirUsuario() y listarUsuarios().	Alta	Sprint 1 (12/Oct)
E2	**Envío y almacenamiento de mensajes**	Gestionar el envío de mensajes entre usuarios y guardarlos en XML.	Alta	Sprint 1 (12/Oct)
HU-2	Como usuario, quiero enviar mensajes a otro usuario.	- Crear clase Mensaje. - Crear mensajes.xml con JAXB. - Implementar enviarMensaje().	Alta	Sprint 1 (12/Oct)

E3	**Visualización de conversaciones**	Permitir ver conversaciones completas entre dos usuarios.	Alta	Sprint 2 (13/Oct)
HU-3	Como usuario, quiero ver mi conversación con otro usuario.	- Implementar verConversacion(). - Leer y filtrar mensajes del XML. - Mostrar en la interfaz.	Alta	Sprint 2 (13/Oct)
E4	**Análisis con Streams**	Generar resúmenes e informes de conversación con Streams.	Media	Sprint 2 (13/Oct)
HU-4	Como usuario, quiero obtener estadísticas de mi conversación.	- Contar mensajes, palabras más usadas, etc. - Mostrar resumen.	Media	Sprint 2 (13/Oct)
E5	**Exportación de conversaciones**	Exportar conversaciones a TXT o CSV con Streams.	Media	Sprint 3 (14/Oct)

HU-5	Como usuario, quiero exportar mis conversaciones a texto.	- Crear método exportarConversacion(). - Escribir en un archivo .txt.	Media	Sprint 3 (14/Oct)
E6	**Archivos adjuntos**	Permitir enviar y visualizar archivos adjuntos.	Media	Sprint 3 (**15/Oct**)
HU-6	Como usuario, quiero enviar archivos adjuntos.	- Guardar archivos en /media/ con InputStream/OutputStream. - Asociar adjunto en el XML. - Permitir abrir adjuntos desde la UI.	Media	Sprint 3 (**15/Oct**)
E7	**Implementación de Logs**	Registrar eventos y errores de la aplicación en un archivo.	Media	Sprint 3 (**16/Oct**)
HU-7	Como desarrollador, quiero un log para depurar la aplicación.	- Configurar java.util.logging para escribir en chat.log. - Añadir registros en puntos clave.	Media	Sprint 3 (**16/Oct**)

E8	**Exportación ZIP (opcional)**	Permitir empaquetar conversaciones y adjuntos en ZIP.	Baja	Sprint 3 (**17/Oct**)
HU-8	Como usuario, quiero exportar todo en un ZIP.	- Implementar ZipOutputStream. - Añadir el .txt y los archivos adjuntos al ZIP.	Baja	Sprint 3 (**17/Oct**)
E9	**Chat online con Sockets (opcional)**	Añadir un modo online cliente-servidor con sockets TCP.	Baja	Sprint 4 (**18-19/Oct**)
HU-9	Como usuario, quiero comunicarme en tiempo real.	- Crear ServidorChat (ServerSocket y Threads). - Crear ClienteChat (Socket). - Integrar con la lógica de guardado en XML.	Baja	Sprint 4 (**18-19/Oct**)
E10	**Pulido de Interfaz (JavaFX)**	Mejorar la usabilidad y el estilo visual de la aplicación.	Media	Sprint 4 (**20/Oct**)
HU-10	Como usuario, quiero una interfaz pulida y funcional.	- Integrar botones para todas las funciones (ZIP, Online). - Mejorar CSS y layout. - Asegurar que la experiencia es intuitiva.	Media	Sprint 4 (**20/Oct**)

E11	**Pruebas y Documentación Final**	Validar todo el proyecto y preparar la entrega.	Alta	Sprint 5 (**24/Oct**)
HU-11	Como estudiante, necesito entregar un proyecto funcional y documentado.	- Pruebas completas de todos los modos. - Preparar el PDF con el modelo y el enlace a Git. - Preparar la demo para la presentación.	Alta	Sprint 5 (**24/Oct**)
-	**Entrega Final**	Fecha límite para la entrega del proyecto.	Crítica	**26/Oct**

15. Conclusión

El proyecto "**Chat Offline XML**" culmina como una aplicación de escritorio **funcional, robusta y bien estructurada**, desarrollada sobre la plataforma JavaFX. Satisface plenamente los requisitos establecidos para un sistema de chat local con persistencia basada en archivos XML, incorporando funcionalidades avanzadas como la gestión integral de archivos adjuntos (validación, almacenamiento, visualización con miniaturas, apertura y exportación individual) y múltiples formatos de exportación de conversaciones (TXT, CSV, ZIP), además de capacidades de análisis estadístico mediante Streams.

La adopción de patrones de diseño consolidados como **MVC** y **DAO**, junto con una clara separación de responsabilidades a través de **clases de utilidad**, ha resultado en una base de código modular y mantenible. La refactorización, que incluye la extracción de lógica duplicada a métodos auxiliares y la definición de constantes, así como el uso de **la interfaz de usuario** mediante la implementación de un **ListView** con celdas personalizadas (**ChatMessageCell**) para simular **burbujas de chat**, elevan la calidad del proyecto y la experiencia del usuario final. La integración de iconos y la posibilidad de enviar mensajes con la tecla Enter son detalles que contribuyen a una mayor usabilidad.

Si bien la elección de **JAXB y archivos XML** para la persistencia cumple con los requisitos del proyecto, es importante reconocer sus **limitaciones en términos de rendimiento y escalabilidad** al manejar grandes volúmenes de datos, dado que requiere la lectura y escritura de archivos completos para operaciones incrementales. No obstante, dentro de este marco tecnológico, la implementación es correcta y funcional.

Las áreas identificadas para **futuras mejoras**, principalmente la optimización del rendimiento mediante el uso de hilos de fondo (**Task/Service**) y la potencial migración a una base de datos embebida, junto con la adición de nuevas funcionalidades (chat grupal, notificaciones), marcan una hoja de ruta clara para la evolución del proyecto.

En su estado actual, "Chat Offline XML" constituye un **ejemplo sólido y completo** de una aplicación de escritorio desarrollada con JavaFX, demostrando la aplicación efectiva de los conceptos y tecnologías abordados en el ciclo formativo DAM.

16. Ayuda IA

La asistencia mediante herramientas de Inteligencia Artificial (IA) ha demostrado ser un recurso valioso para **acelerar la implementación** y **mejorar la calidad** de ciertas secciones del código, actuando como un asistente de programación avanzado.

16.1. Implementación de FileChooser / DirectoryChooser

La IA contribuyó a generar rápidamente el código necesario para utilizar los diálogos estándar de JavaFX para la selección de archivos y directorios. Esto incluyó:

- La instanciación y configuración básica (**setTitle, setInitialFileName**).
- La correcta definición y aplicación de filtros de extensión (ExtensionFilter).
- El manejo del valor de retorno, considerando el caso en que el usuario cancela la operación (**null**). Estos elementos se aprecian en los métodos **seleccionarAdjunto** y en el método refactorizado **mostrarDialogoExportar** de **MainController**.

16.2. Generación de Archivos ZIP

La creación de archivos ZIP (**FileManager.crearArchivoZip**) implica el manejo de Streams de I/O, una tarea propensa a errores si no se gestionan correctamente los recursos. La IA proporcionó:

- Una implementación **eficiente** basada en **ZipOutputStream** y **FileInputStream**, utilizando un buffer para leer los archivos adjuntos por bloques, lo que optimiza el uso de memoria.
- El uso correcto de **try-with-resources**, garantizando que todos los streams se cierren automáticamente, incluso si ocurren excepciones, previniendo así fugas de recursos.
- La lógica para añadir correctamente las diferentes entradas (**ZipEntry**) al archivo ZIP, tanto para el archivo de texto de la conversación como para cada uno de los archivos adjuntos físicos.

16.3. Implementación de ChatMessageCell (Celda de Mensaje Personalizada)

Esta es, posiblemente, **una de las contribuciones más significativas de la IA** en este proyecto, dada la complejidad relativa de crear **ListCell** personalizadas en JavaFX. La asistencia de la IA fue clave para generar la clase **ChatMessageCell**, incluyendo:

- La **estructura fundamental** de la clase, extendiendo **ListCell<Mensaje>** y sobrescribiendo el método **updateItem**.
- La **composición del layout** interno usando nodos JavaFX (**HBox, VBox, Label, Hyperlink, ImageView**) para construir la apariencia visual de la burbuja de chat.
- La **lógica esencial dentro de updateItem** para:
    - Poblar los **Label** con los datos del **Mensaje** (remitente, texto, hora formateada).
    - **Resolver el problema del ajuste de texto y el ancho de la burbuja**, implementando **setWrapText(true), setMaxWidth(Double.MAX_VALUE)** y, crucialmente, el **binding** de **maxWidthProperty** del **Label** principal al ancho del **ListView** para un comportamiento adaptativo.
    - Gestionar la **diferenciación visual y alineación** de los mensajes propios (**own-message**) frente a los de otros usuarios (**other-message**) mediante clases CSS y **setAlignment**.
    - Implementar la **visualización condicional de adjuntos**, mostrando un **Hyperlink** clicable (con acción para abrir el archivo) y una miniatura (**ImageView**) en caso de ser una imagen.
- La configuración del **cellFactory** en **MainController** para integrar esta celda personalizada con el **ListView** del chat.

16.4. Implementación de StreamUtils para Estadísticas

La clase StreamUtils, que calcula las estadísticas del chat, hace un uso intensivo de la API de Streams de Java. La IA probablemente asistió en:

- **La formulación de las operaciones de Stream** (filter, map, flatMap, groupingBy, counting, sorted, limit, collect) para calcular eficientemente:
    - El conteo de mensajes por usuario (contarMensajesPorUsuario).
    - La identificación y conteo de las palabras más comunes (palabraMasComun), incluyendo pasos de normalización (minúsculas) y filtrado (longitud).
- **La implementación de los colectores** (Collectors.groupingBy, Collectors.counting, Collectors.toMap con LinkedHashMap para preservar orden).
- La generación de los **métodos de formateo** (formatearConteoUsuario, formatearTopPalabras) que convierten los Map resultantes en String elegibles para la interfaz o exportación.

16.5. Versión Online (Sockets)

*(Esta funcionalidad se encuentra en la rama **Version-online**)*

La versión online experimental utiliza **Sockets** de Java (**java.net.ServerSocket, java.net.Socket**) para establecer una comunicación cliente-servidor básica:

- **Servidor (Servidor.java):**
    - Escucha en un puerto definido (**12345**).
    - Utiliza un **ExecutorService** (pool de hilos) para manejar múltiples clientes concurrentemente.
    - Cada cliente conectado es gestionado por una instancia de **ClientHandler** en su propio hilo.
    - Mantiene un conjunto (**Set**) sincronizado de **PrintWriter** para retransmitir (**broadcast**) los mensajes recibidos a todos los demás clientes conectados.
- **Cliente (Cliente.java):**
    - Se conecta al servidor en la dirección y puerto especificados (**localhost:12345**).
    - Utiliza **JAXB** para serializar objetos **Mensaje** a XML antes de enviarlos por el socket y para deserializar el XML recibido del servidor.
    - Inicia un **hilo separado (listenerThread)** para escuchar continuamente los mensajes entrantes del servidor sin bloquear la interfaz de usuario.
    - Utiliza el patrón **callback** (**Consumer<Mensaje> onMessageReceived, Consumer<EstadoRed> onNetworkStateChanged**) para notificar al **MainController** sobre:
        - Mensajes nuevos recibidos (que se procesan en el hilo de JavaFX mediante **Platform.runLater**).
        - Cambios en el estado de la conexión (Conectado, Desconectado, Error), permitiendo actualizar la UI (**lblEstadoRed**).
- **Integración (MainController.java):**
    - En **initialize**, se crea una instancia del **Cliente**, pasándole los *callbacks* necesarios.
    - El método **enviarMensaje** ahora serializa el **Mensaje** a XML y lo envía a través del **Cliente**, además de guardarlo localmente en el XML (para mantener la persistencia offline).
    - Se implementa **recibirMensajeDelServidor** para procesar los mensajes que llegan del *callback* del **Cliente** y actualizar el **ListView** del chat.
    - Se añade **actualizarEstadoRed** para mostrar el estado de la conexión en la interfaz.