



Universidad
de Alcalá

PRÁCTICA PARTE 2:

“LINK UP”

RED SOCIAL SENCILLA

Alejandro Darío Albert Calugar

Daniel Senespleda Fernández

INGENIERÍA DE COMPUTADORES CURSO 2024/25

Tabla de contenido

Diagrama E/R	3
Diagrama de clases	3
Diagrama E/R	3
API	4
Red Social (Link up)	4
Objetivos del proyecto	4
Desarrollo Backend	5
JSP	5
index.html	5
admin.jsp	5
login.jsp	6
messages.jsp	6
profile.jsp	7
register.jsp	8
search.jsp	9
timeline.jsp	9
Java Servlets	10
AceptarSolicitudServlet.java	10
AdminServlet.java	10
Admin_PublicacionDAO.java	11
Admin_UsuarioDAO.java	11
AuthServlet.java	12
Comentario.java	13
ComentarioDAO.java	13
FriendRequestDAO.java	14
LikeDAO.java	16
Mensaje.java	16
MensajeDAO.java	16
MessagesServlet.java	17
ProfileServlet.java	17
Publicacion.java	18
PublicacionDAO.java	18
RechazarSolicitudServlet.java	19
SearchServlet.java	19
SolicitudAmistad.java	20
TimelineServlet.java	20
Usuario.java	20
UsuarioDAO.java	20
Creación de tablas para la Base de Datos	21
Código SQL:	21
Inserción de datos de prueba	23
Base de Datos	24
Patrón MVC	24
Modelo (Model)	24
Ejemplo de clases del modelo:	24
Vista (View)	24
Ejemplo de archivos JSP:	25
Controlador (Controller)	25
Ejemplo de Servlets:	25

Diagrama E/R

Diagrama de clases

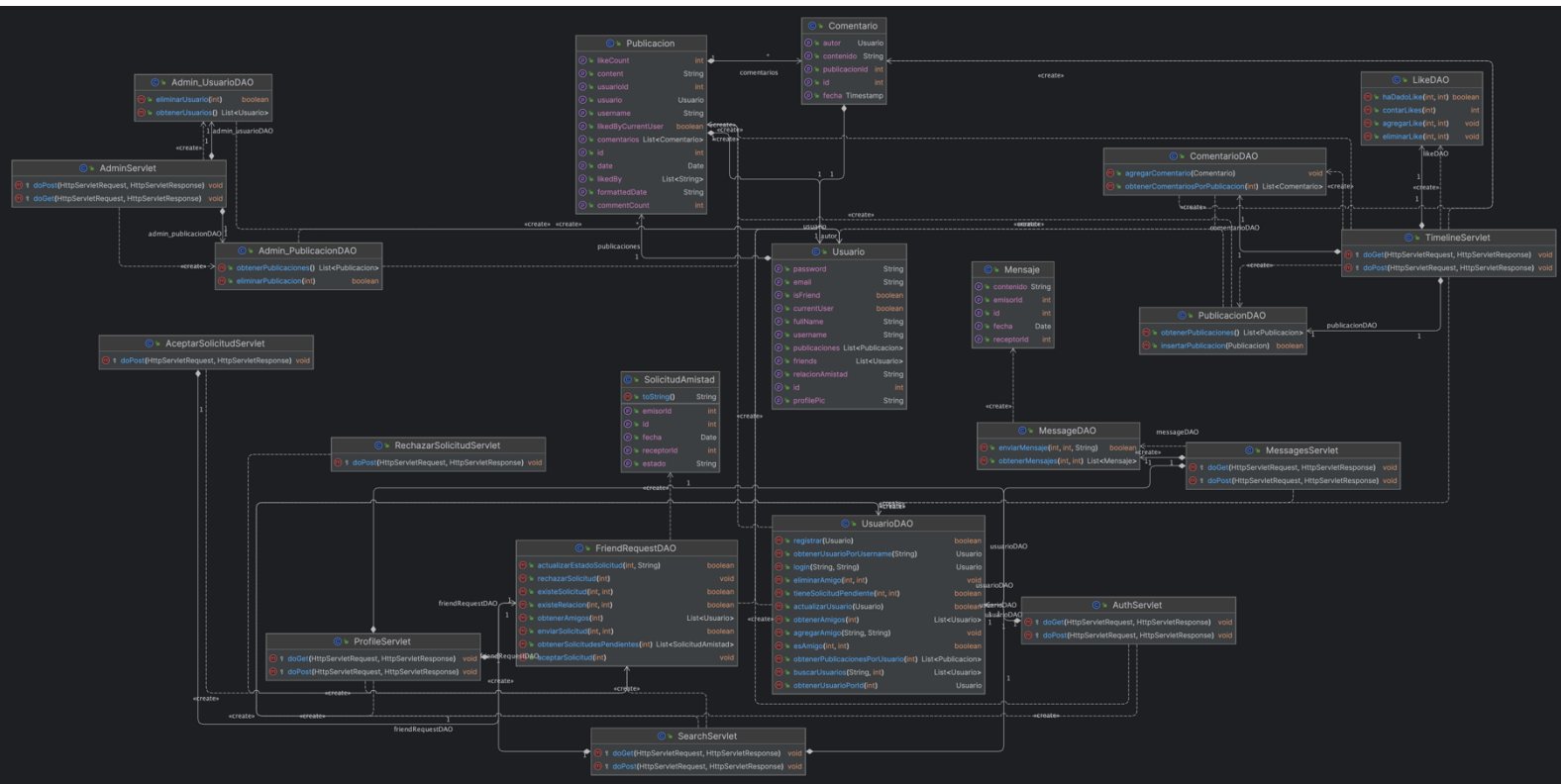
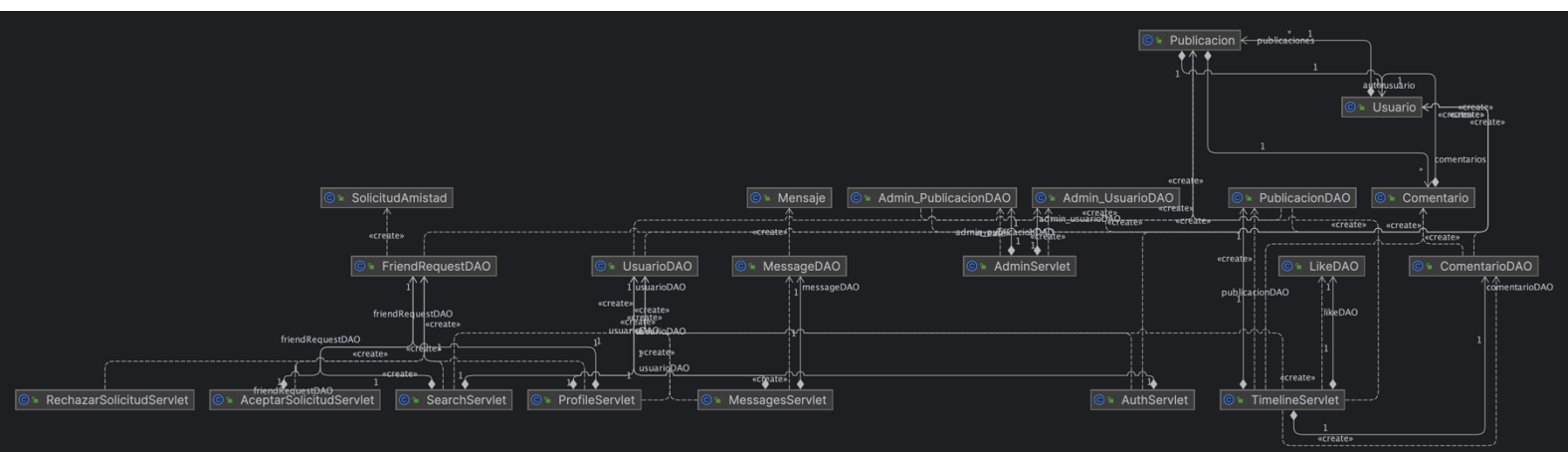


Diagrama E/R



API

Acción	Descripción	Ruta de acceso	Método HTTP	Parámetros de entrada	Respuesta
Obtener todas las publicaciones	Recuperar todas las publicaciones disponibles	/api/publicaciones	GET	Ninguno	Lista de publicaciones
Obtener una publicación	Recupera una publicación específica por su ID	/api/publicaciones/{id}	GET	id	Detalles de la publicación
Crear una nueva publicación	Crear una nueva publicación	/api/publicaciones	POST	content, usuarioID	Detalles de la publicación creada
Actualizar una nueva publicación	Actualiza una publicación existente	/api/publicaciones/{id}	PUT	id, content	Detalles de la publicación actualizada
Eliminar una publicación	Elimina una publicación existente	/api/publicaciones/{id}	DELETE	id	Mensaje de éxito o error
obtener todos los usuarios	Recupera todos los usuarios registrados	/api/usuarios	GET	Ninguno	Lista de usuarios
Obtener un usuario	Recupera un usuario específico por su ID	/api/usuarios/{id}	GET	id	Detalles del usuarios
Crear un nuevo usuario	Crea un nuevo usuario	/api/usuarios	POST	fullName, email, username, password	Detalles del usuario creado
Actualizar un usuario	Actualiza un usuario existente	/api/usuarios/{id}	PUT	id, fullName, email, username, password	Detalles del usuario actualizado
Eliminar un usuario	Elimina un usuario existente	/api/usuarios/{id}	DELETE	id	Mensaje de éxito o error
Obtener comentarios de una publicación	Recupera todos los comentarios de una publicación	/api/publicaciones/{id}/comentarios	GET	id	Lista de comentarios
Crear un comentario	Crea un nuevo comentario en una publicación	/api/publicaciones/{id}/comentarios	POST	id, contenido	Detalles del comentario creado
Obtener solicitudes de amistad	Recupera todas las solicitudes de amistad pendientes	/api/solicitudes	GET	Ninguno	Lista de solicitudes de amistad
Enviar solicitud de amistad	Envía una solicitud de amistad a otro usuario	/api/solicitudes	POST	emisorId, receptorId	Detalles de la solicitud de amistad
Aceptar solicitud de amistad	Acepta una solicitud de amistad	/api/solicitudes/{id}/aceptar	POST	id	Mensaje de éxito o error
Rechazar solicitud de amistad	Rechaza una solicitud de amistad	/api/solicitudes/{id}/rechazar	POST	id	Mensaje de éxito o error
Inicio de sesión	Establece la sesión del usuario	/usu/RedSocial/login	POST	username, password	Redirige a la página de perfil o muestra mensaje de error
Cerrar sesión	Cierra la sesión del usuario	/usu/RedSocial/profile	GET	(N/A)	Redirige a la página de inicio
Ver perfil	Permite ver los datos del usuario	/usu/RedSocial/profile	POST		Redirige a la página de perfil del usuario

Red Social (Link up)

Objetivos del proyecto

El objetivo principal de esta práctica es seguir desarrollando Link Up, una red social sencilla e interactiva, centrando los esfuerzos en ampliar y mejorar la funcionalidad del Backend. Mientras que en la primera práctica se utilizó LocalStorage para gestionar y persistir los datos de la aplicación en el lado cliente, esta segunda práctica se enfocará en trasladar dicha gestión al servidor para garantizar mayor escalabilidad, seguridad y robustez.

Para ello, implementaremos tecnologías como Java Servlets y Java Server Pages (JSP). Los Java Servlets se usarán para procesar las peticiones y respuestas entre el cliente y el servidor, gestionando la lógica que antes residía en el cliente. Por su parte, las JSP permitirán generar contenido dinámico en las páginas web, integrando las respuestas del servidor con una interfaz más rica y funcional.

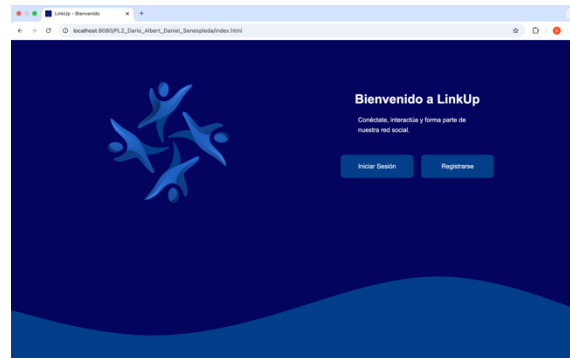
En la primera práctica, el Backend se implementó de forma básica utilizando LocalStorage, lo que permitió almacenar y gestionar datos como usuarios, publicaciones y comentarios directamente en el navegador del cliente. Aunque este enfoque fue funcional para una versión inicial, en esta segunda etapa se busca trasladar estas capacidades al servidor. Esto garantizará que los datos sean compartidos y sincronizados entre diferentes usuarios, además de proporcionar una base más sólida para futuras ampliaciones y mejoras.

Desarrollo Backend

JSP

index.html

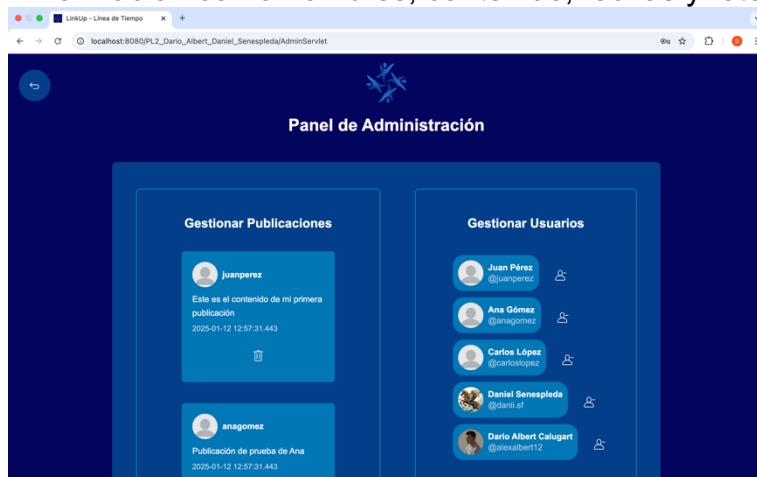
Este código HTML es la estructura de la página de bienvenida de la red social "LinkUp". En la parte lógica, se presenta un encabezado con el logo de la aplicación y un mensaje de bienvenida, que invita al usuario a registrarse o iniciar sesión.



Los dos botones, "Iniciar Sesión" y "Registrarse", dirigen a las páginas correspondientes: login.jsp y register.jsp. Además, se incluye un SVG al final de la página para crear un fondo animado en forma de ola.

admin.jsp

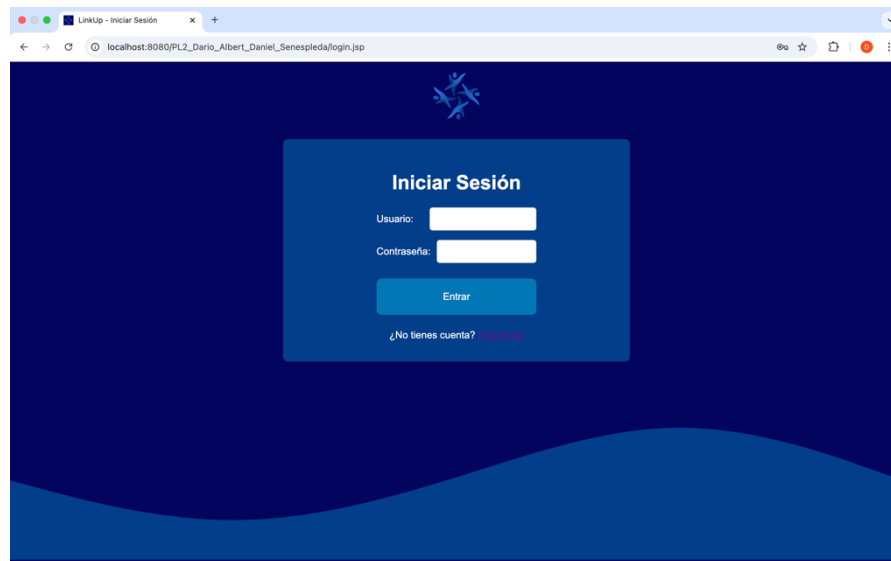
Con esta clase hemos creado un panel de administración para la red social LinkUp, donde los administradores pueden ver y gestionar publicaciones y usuarios. Utiliza JSP y JSTL para mostrar las publicaciones y usuarios de forma dinámica, con bucles que recorren las listas de publicaciones y usuarios para mostrar información como nombres, contenido, fechas y fotos de perfil.



También incluye formularios que permiten eliminar publicaciones y usuarios enviando los datos al servlet AdminServlet. Todo está diseñado con HTML y CSS, y además tiene un botón para cerrar sesión que lleva a la página principal.

login.jsp

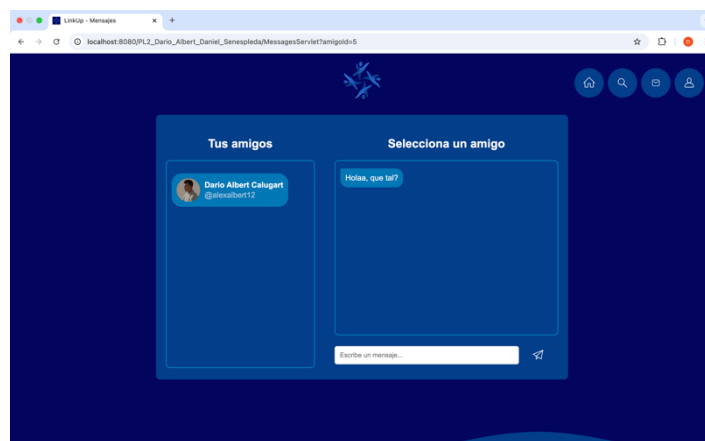
Este código es la página de inicio de sesión.



- **Formulario de Inicio de Sesión:** El formulario recoge el nombre de usuario y contraseña del usuario. Los datos se envían al Servlet AuthServicelet mediante el método POST para procesar la autenticación.
- **Campo oculto:** Se incluye un campo oculto con el nombre action y el valor login para que el servlet pueda identificar la acción que debe ejecutar.
- **Validación:** Los campos de usuario y contraseña son obligatorios gracias al atributo required. Si el usuario deja uno de estos campos vacío, el formulario no se enviará.
- **Manejo de Errores:** Si se produce un error durante el inicio de sesión, el mensaje de error se muestra debajo del formulario, utilizando la variable de expresión \${error} que se espera que sea pasada desde el servlet.
- **Redirección a Registro:** Si el usuario aún no tiene cuenta, puede hacer clic en el enlace que lo llevará a la página de registro (register.jsp).
- **Animación de Fondo:** Se incluye un SVG que genera una ola animada en la parte inferior de la página.

messages.jsp

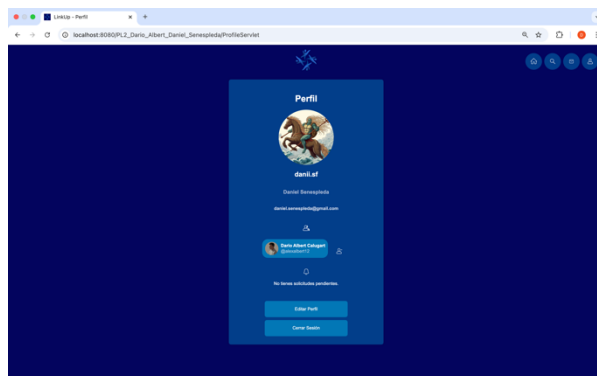
Este código implementa la página de mensajes:



- **Vista de Amigos:** En la barra lateral, se muestra una lista de los amigos del usuario. Se recorre la lista de amigos utilizando JSTL (<c:forEach>), y por cada amigo se genera un enlace que permite al usuario seleccionar un amigo para iniciar una conversación. Al hacer clic en el nombre de un amigo, se envía una solicitud al MessagesServlet con el amigoid del amigo seleccionado.
- **Vista de Mensajes:** En la sección principal de la página, se despliega un área de chat donde se muestran los mensajes entre el usuario y el amigo seleccionado. Los mensajes se recorren de la misma manera que la lista de amigos, utilizando JSTL para mostrar el contenido de cada mensaje. Se diferencia el mensaje enviado por el usuario (message-sent) y los mensajes recibidos (message-received), mediante clases CSS para aplicar estilos diferentes.
- **Formulario de Envío de Mensajes:** El formulario permite al usuario escribir un mensaje y enviarlo al amigo seleccionado. El formulario tiene un campo oculto (receptorId) que contiene el ID del amigo seleccionado para que el servlet pueda procesar correctamente el destinatario del mensaje. El contenido del mensaje es obligatorio (required).
- **Botones de Navegación:** En la parte inferior de la página, se encuentran botones para navegar a otras secciones de la plataforma:
 - o **Casa:** Redirige al timeline de publicaciones.
 - o **Buscar Amigos:** Redirige a la página de búsqueda de amigos.
 - o **Mensajes:** Recarga la vista de mensajes.
 - o **Perfil:** Redirige al perfil del usuario.

profile.jsp

Este código muestra el perfil de un usuario en una red social. Incluye varias secciones clave:



- **Encabezado:** Muestra el logo de la página.
- **Perfil del usuario:** Muestra la foto, nombre, y correo electrónico del usuario usando los atributos pasados desde el servidor (request.getAttribute()).
- **Lista de amigos:** Si el usuario tiene amigos, se muestran sus fotos y nombres, con la opción de eliminar un amigo. Si no tiene amigos, se muestra un mensaje diciendo "No tienes amigos aún".
- **Solicitudes pendientes:** Si hay solicitudes de amistad pendientes, se muestran con botones para aceptar o rechazar. Si no hay solicitudes, también se muestra un mensaje correspondiente.

- **Botones de acción:** Un botón para editar el perfil, que revela un formulario con campos para cambiar el nombre, correo electrónico y foto de perfil. Un botón para cerrar sesión.
- **Formulario de edición de perfil:** Permite al usuario modificar su nombre completo, correo electrónico y subir una nueva foto de perfil. Incluye botones para guardar cambios o cancelar.

register.jsp

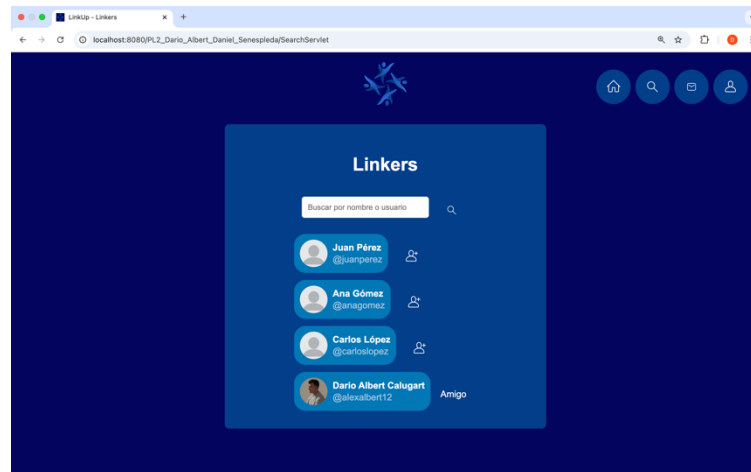
Este código es una página JSP para el registro de usuarios

- **Importación de librerías:** Se importan clases de Java necesarias para conectarse a la base de datos (como DriverManager, Connection, PreparedStatement, y ResultSet), aunque no se utilizan directamente en este archivo JSP. Esto probablemente se utilizará en el servlet que maneja la lógica de registro.
- **Encabezado HTML:** Contiene información básica como el título de la página ("LinkUp - Registrarse"), la vinculación a una hoja de estilo CSS (style.css) y el ícono de la página (LinkUp.ico).

- **Formulario de Registro:** Permite al usuario ingresar su nombre completo, correo electrónico, nombre de usuario y contraseña. Cada campo tiene una etiqueta (label) que indica qué debe ingresar el usuario, y cada campo de entrada tiene un atributo required para que no se envíe el formulario vacío. El formulario envía una solicitud POST al servlet AuthServlet, con una acción de "register" para realizar el registro del usuario.
- **Mensaje de error:** Si ocurre algún error durante el registro (como un correo ya registrado), se muestra un mensaje de error en color rojo, que se pasa desde el servidor como un atributo \${error}.
- **Enlace para iniciar sesión:** Si el usuario ya tiene cuenta, puede hacer clic en un enlace para ir a la página de inicio de sesión.

search.jsp

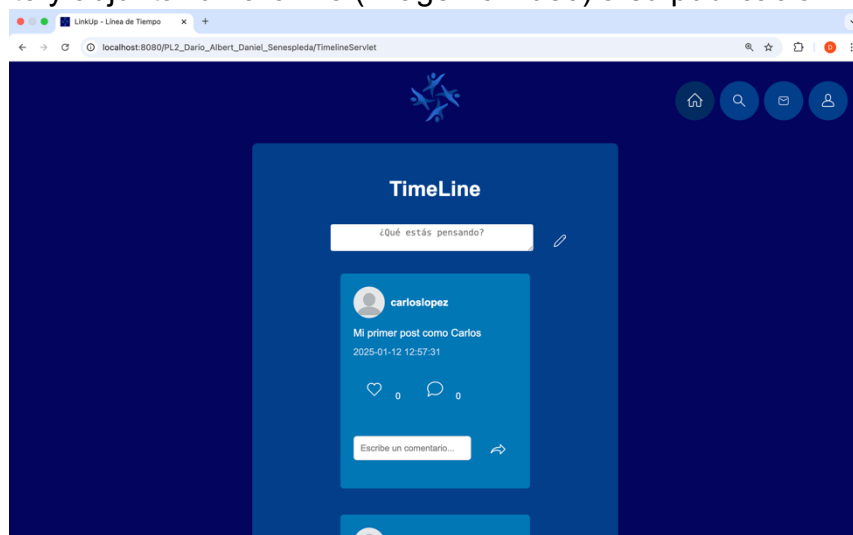
Se trata de la clase sobre la búsqueda de usuarios en la red social LinkUp. Permite al usuario buscar por nombre o nombre de usuario. Muestra los resultados de la búsqueda con los datos de cada usuario (nombre, foto de perfil, y nombre de usuario), y según la relación de amistad (amigos, pendiente o disponible), ofrece la opción de enviar una solicitud de amistad. Además, incluye botones de navegación para otras secciones como la línea de tiempo, mensajes y perfil. La página tiene un diseño responsive y moderno, con botones e íconos SVG para mejorar la interfaz.



timeline.jsp

Esta clase muestra una línea de tiempo con publicaciones y comentarios. Está estructurado de la siguiente manera:

- **Formulario para crear publicaciones:** Permite al usuario escribir un texto y adjuntar un archivo (imagen o video) a su publicación.



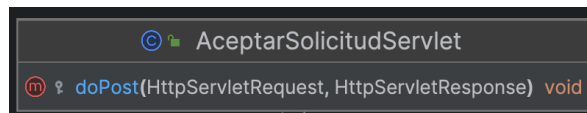
- **Publicaciones:** Utiliza JSTL (<c:forEach>) para iterar sobre una lista de publicaciones (publicaciones). Cada publicación contiene:
 - Un campo con el nombre de usuario y su foto de perfil.
 - El contenido de la publicación.
 - La fecha de la publicación.

- Un botón para dar "me gusta", que cambia dependiendo de si el usuario ya dio "me gusta".
 - Un contador de "me gusta".
 - Un botón para agregar comentarios, con un contador de comentarios y un formulario para enviarlos.
 - Si no hay publicaciones, se muestra un mensaje informando de ello.
- **Menú de navegación:** Botones para navegar entre las diferentes secciones de la aplicación (Inicio, Búsqueda, Mensajes y Perfil).

Java Servlets

AceptarSolicitudServlet.java

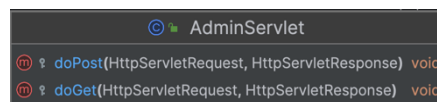
El servlet maneja una solicitud POST para aceptar una solicitud de amistad y agregarla a la lista de amigos.



- **Recibe una solicitud de amistad:** Extrae el solicitudId del parámetro de la solicitud HTTP.
- **Validación:** Si el solicitudId es nulo o vacío, redirige al usuario a la página de perfil sin hacer nada más.
- **Procesa la solicitud:** Si el solicitudId es válido, intenta marcar la solicitud como aceptada y agregarla a la lista de amigos mediante el método aceptarSolicitud de FriendRequestDAO.
- **Manejo de errores:** Si ocurre un error al aceptar la solicitud (por ejemplo, un problema con la base de datos), se captura la excepción SQLException y se establece un mensaje de error.
- **Redirección:** Después de procesar la solicitud, redirige al usuario nuevamente a la página de perfil.

AdminServlet.java

El servlet maneja las solicitudes GET y POST para gestionar publicaciones y usuarios en el panel de administración.



- **Método doGet:** Obtiene todas las publicaciones y usuarios desde las bases de datos utilizando los DAOs Admin_PublicacionDAO y Admin_UsuarioDAO. Pasa las listas de publicaciones y usuarios al JSP (admin.jsp) para que se muestren en la interfaz de administración. Redirige a la página admin.jsp, que probablemente mostrará los datos de los usuarios y las publicaciones.
- **Método doPost:** Recibe una acción desde el formulario (action), que puede ser eliminar una publicación o un usuario. Si la acción es eliminarPublicacion,

obtiene el `publicacionId` y llama al método `eliminarPublicacion` de `Admin_PublicacionDAO`. Si la acción es `eliminarUsuario`, obtiene el `usuarioId` y llama al método `eliminarUsuario` de `Admin_UsuarioDAO`. Después de realizar la acción, redirige al mismo servlet (`AdminServlet`) para refrescar la página.

Admin_PublicacionDAO.java

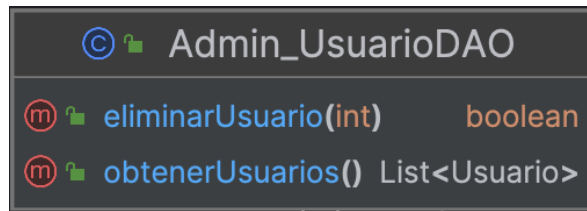
La clase interactúa con la base de datos para obtener y eliminar publicaciones, así como los datos relacionados.



- **Método `obtenerPublicaciones`:** Obtiene todas las publicaciones desde la base de datos. Su **funcionamiento** es el siguiente:
 - o Realiza una consulta SQL que une las tablas `PUBLICACION` y `USUARIO` para obtener información sobre las publicaciones (`ID`, `contenido`, `fecha`) y sobre el usuario (`ID`, `nombre de usuario`, `nombre completo`, `foto de perfil`).
 - o Crea un objeto `Publicacion` y asigna los valores obtenidos de la consulta.
 - o Asocia un objeto `Usuario` a cada publicación.
 - o Añade cada publicación a una lista y la devuelve al final.
- **Método `eliminarPublicacion`:** Elimina una publicación junto con sus elementos relacionados (`likes` y `comentarios`). Su funcionamiento es:
 - o Utiliza tres consultas SQL para eliminar los `likes`, los `comentarios` y la publicación en sí.
 - o Desactiva el `autocommit` para poder realizar la eliminación de manera transaccional, garantizando que si algo falla, se reviertan todos los cambios.
 - o Ejecuta cada eliminación en orden: primero los `likes`, luego los `comentarios`, y finalmente la publicación.
 - o Si todas las operaciones se realizan correctamente, se confirma la transacción y se retorna `true`. Si ocurre un error, la transacción se revierte y el método retorna `false`.

Admin_UsuarioDAO.java

La clase interactúa con la base de datos para obtener y eliminar usuarios, así como los datos relacionados.

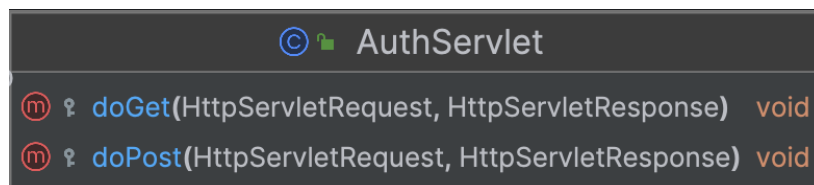


- **Método obtenerUsuarios:** Obtiene todos los usuarios desde la base de datos. Su funcionamiento es el siguiente:
 - Realiza una consulta SQL para obtener todos los datos de la tabla USUARIO.
 - Crea un objeto Usuario y asigna los valores obtenidos de la consulta (ID, nombre de usuario, nombre completo, correo electrónico, foto de perfil).
 - Añade cada usuario a una lista y la devuelve al final.
- **Método eliminarUsuario:** Elimina un usuario junto con sus elementos relacionados (mensajes, relaciones de amistad y solicitudes de amistad). Su funcionamiento es:
 - Utiliza cuatro consultas SQL para eliminar los mensajes, las relaciones de amistad, las solicitudes de amistad y, finalmente, el usuario en sí.
 - Desactiva el autocommit para poder realizar la eliminación de manera transaccional, garantizando que, si algo falla, se reviertan todos los cambios.
 - Ejecuta cada eliminación en orden: primero los mensajes, luego las relaciones de amistad, después las solicitudes de amistad, y finalmente el usuario.

Si todas las operaciones se realizan correctamente, se confirma la transacción y se retorna true. Si ocurre un error, la transacción se revierte y el método retorna false.

AuthServlet.java

La clase interactúa con el proceso de autenticación de los usuarios en la aplicación web.



- **Método doPost:** Gestiona las solicitudes de login y registro. Su funcionamiento es el siguiente:
 - Verifica si la acción solicitada es un **login** o un **registro**.
 - **Login:**
 - Si las credenciales corresponden al usuario administrador, se autentica al administrador y se redirige al servlet de administración (AdminServlet).

- Si las credenciales corresponden a un usuario válido, se autentica al usuario y se redirige al servlet de perfil (ProfileServlet).
 - Si las credenciales son inválidas, se muestra un mensaje de error y se redirige a la página de login.
- **Registro:**
 - Si la acción es un registro, crea un nuevo usuario con los datos proporcionados y una imagen de perfil predeterminada.
 - Si el registro es exitoso, se muestra un mensaje de éxito y se redirige a la página de login.
 - Si el usuario ya existe, se muestra un mensaje de error y se redirige a la página de registro.
- **Método doGet:** Este método responde a las solicitudes GET con un mensaje simple para verificar que el servlet está funcionando correctamente.

Comentario.java

La clase representa un **Comentario** realizado por un usuario en una publicación.



• Atributos:

- id: Identificador único del comentario.
- publicacionId: Identificador de la publicación a la que pertenece el comentario.
- autor: Objeto de tipo Usuario que representa al usuario que realizó el comentario.
- contenido: Texto que contiene el contenido del comentario.
- fecha: Fecha y hora en que el comentario fue realizado, representada por un objeto Timestamp.
-

• Métodos:

- **Getters y Setters:** Permiten acceder y modificar los valores de los atributos definidos en la clase. Estos métodos facilitan la interacción con los objetos de tipo Comentario, permitiendo recuperar y actualizar la información sobre el comentario y sus atributos.

ComentarioDAO.java

Es una clase encargada de interactuar con la base de datos para obtener y agregar comentarios en el sistema. Esta clase facilita las operaciones de consulta y modificación relacionadas con los comentarios de las publicaciones.



- **Método obtenerComentariosPorPublicacion:** Obtiene todos los comentarios asociados a una publicación específica.
 - **Funcionamiento:**
 - Realiza una consulta SQL que une las tablas COMENTARIOS y USUARIO para obtener información sobre el comentario (ID, contenido, fecha) y sobre el usuario (ID, nombre de usuario, nombre completo, foto de perfil).
 - Crea un objeto Comentario para cada fila del resultado de la consulta y asigna los valores obtenidos de la base de datos.
 - Asocia un objeto Usuario a cada comentario para representar al autor.
 - Agrega cada comentario a una lista de comentarios y la devuelve al final.
- **Método agregarComentario:** Inserta un nuevo comentario en la base de datos.
 - **Funcionamiento:**
 - Ejecuta una consulta SQL para insertar un nuevo comentario en la tabla COMENTARIOS, incluyendo los datos sobre la publicación, el usuario (autor), el contenido y la fecha del comentario.
 - Utiliza un PreparedStatement para evitar inyecciones SQL y garantizar la seguridad de los datos.

FriendRequestDAO.java

Es una clase encargada de gestionar las solicitudes de amistad y las relaciones entre usuarios dentro del sistema de red social. Esta clase interactúa con la base de datos para realizar las operaciones necesarias en cuanto a solicitudes de amistad, relaciones de amistad y el seguimiento de estas acciones.

© FriendRequestDAO		
Ⓜ	actualizarEstadoSolicitud(int, String)	boolean
Ⓜ	rechazarSolicitud(int)	void
Ⓜ	existeSolicitud(int, int)	boolean
Ⓜ	existeRelacion(int, int)	boolean
Ⓜ	obtenerAmigos(int)	List<Usuario>
Ⓜ	enviarSolicitud(int, int)	boolean
Ⓜ	obtenerSolicitudesPendientes(int)	List<SolicitudAmistad>
Ⓜ	aceptarSolicitud(int)	void

1. **enviarSolicitud:**
 - Este método permite que un usuario envíe una solicitud de amistad a otro.
 - Realiza una inserción en la tabla SOLICITUDES_AMISTAD con los IDs del emisor y receptor.

- Si la operación tiene éxito, retorna true, de lo contrario, maneja cualquier error y devuelve false.
- 2. **existeRelacion:**
 - Verifica si dos usuarios ya son amigos, consultando la tabla AMIGOS.
 - Retorna true si ya existe una relación de amistad entre ambos usuarios, y false si no.
- 3. **existeSolicitud:**
 - Comprueba si ya existe una solicitud pendiente entre dos usuarios. Esto se hace con una consulta a la tabla SOLICITUDES_AMISTAD.
 - Si ya existe una solicitud pendiente, retorna true, y false en caso contrario.
- 4. **obtenerSolicitudesPendientes:**
 - Obtiene todas las solicitudes de amistad pendientes de un receptor en particular.
 - Devuelve una lista de objetos SolicitudAmistad que contienen la información sobre las solicitudes pendientes.
- 5. **actualizarEstadoSolicitud:**
 - Permite actualizar el estado de una solicitud de amistad (por ejemplo, a "aceptada" o "rechazada").
 - Realiza una actualización en la tabla SOLICITUDES_AMISTAD con el nuevo estado de la solicitud.
- 6. **aceptarSolicitud:**
 - Este método acepta una solicitud de amistad, lo que implica insertar una nueva relación de amistad en la tabla AMIGOS y actualizar el estado de la solicitud en la tabla SOLICITUDES_AMISTAD.
 - Utiliza una transacción para asegurar que ambas operaciones (inserción y actualización) se realicen correctamente o, en caso de error, se reviertan.
- 7. **rechazarSolicitud:**
 - Rechaza una solicitud de amistad eliminándola de la tabla SOLICITUDES_AMISTAD.
 - Si el proceso es exitoso, la solicitud se elimina permanentemente.
- 8. **obtenerAmigos:**
 - Este método obtiene todos los amigos de un usuario específico. Consulta la tabla AMIGOS para recuperar los usuarios relacionados y obtener su información.
 - Devuelve una lista de objetos Usuario correspondientes a los amigos del usuario especificado.

LikeDAO.java

La clase gestiona los "likes" en las publicaciones de una red social. Se conecta a una base de datos Derby y proporciona los siguientes métodos:



```
© LikeDAO
(m) haDadoLike(int, int) boolean
(m) contarLikes(int) int
(m) agregarLike(int, int) void
(m) eliminarLike(int, int) void
```

1. **contarLikes(int publicacionId):** Retorna el número de "likes" de una publicación.

2. **haDadoLike(int publicacionId, int usuarioid):** Verifica si un usuario ha dado "like" a una publicación.

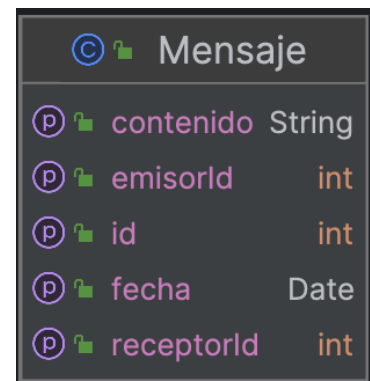
3. **agregarLike(int publicacionId, int usuarioid):** Agrega un "like" de un usuario a una publicación.

4. **eliminarLike(int publicacionId, int usuarioid):** Elimina el "like" de un usuario en una publicación.

Mensaje.java

Representa un mensaje entre dos usuarios en la red social.

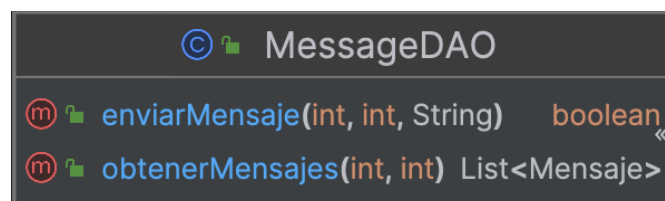
- **Atributos:**
 - id: Identificador del mensaje.
 - emisorId: ID del usuario que envía el mensaje.
 - receptorId: ID del usuario que recibe el mensaje.
 - contenido: Texto del mensaje.
 - fecha: Fecha y hora de envío.
- **Métodos:** Getters y setters para cada atributo.



```
© Mensaje
(p) contenido String
(p) emisorId int
(p) id int
(p) fecha Date
(p) receptorId int
```

MensajeDAO.java

Gestiona la persistencia de los mensajes en la base de datos.



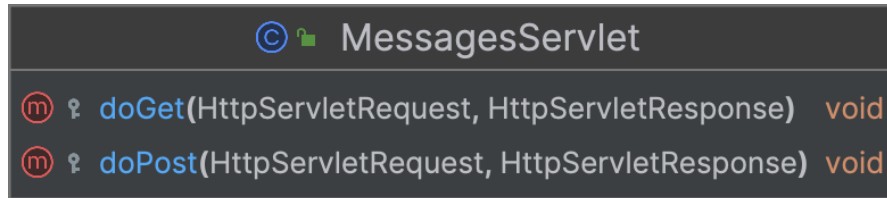
```
© MessageDAO
(m) enviarMensaje(int, int, String) boolean
(m) obtenerMensajes(int, int) List<Mensaje>
```

- **Métodos:**
 - **enviarMensaje(int emisorId, int receptorId, String contenido):** Inserta un nuevo mensaje en la base de datos.
 - **obtenerMensajes(int usuarioid1, int usuarioid2):** Recupera los mensajes entre dos usuarios específicos, ordenados por fecha.

Conexión a la base de datos Derby y ejecución de consultas SQL.

MessagesServlet.java

Controlador de los mensajes entre usuarios en la red social.



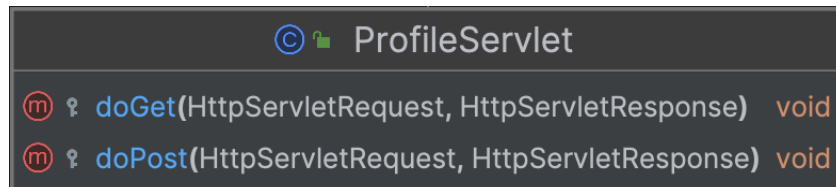
- **Métodos:**

- **doGet(HttpServletRequest request, HttpServletResponse response):** Muestra los mensajes entre el usuario actual y otro usuario, obteniendo los amigos y mensajes.
- **doPost(HttpServletRequest request, HttpServletResponse response):** Permite enviar un mensaje de un usuario a otro. Si el mensaje se envía correctamente, redirige a la vista de mensajes.

Gestiona la interacción con los datos de usuario y los mensajes, utilizando UsuarioDAO y MessageDAO.

ProfileServlet.java

Controla el perfil de usuario en la red social, incluyendo la actualización de datos y la gestión de amigos.



- **Métodos:**

- **doGet(HttpServletRequest request, HttpServletResponse response):** Muestra la información del perfil del usuario, incluyendo amigos y solicitudes de amistad pendientes.
- **doPost(HttpServletRequest request, HttpServletResponse response):** Procesa las acciones relacionadas con el perfil:
 - **Editar perfil:** Permite actualizar el nombre completo, correo y foto de perfil.
 - **Eliminar amigo:** Elimina a un amigo de la lista del usuario.
 - **Aceptar/Rechazar solicitud de amistad:** Actualiza el estado de las solicitudes de amistad.

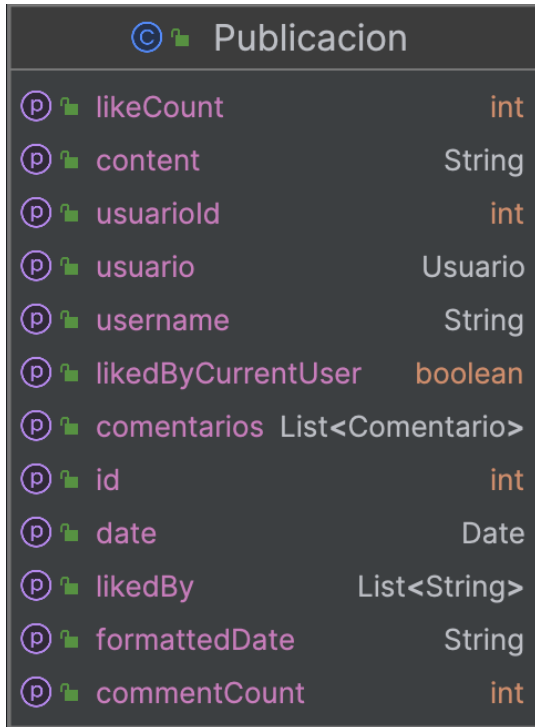
Manejo de archivos subidos para la foto de perfil y actualización de la base de datos mediante UsuarioDAO y FriendRequestDAO.

Publicacion.java

Esta clase representa una publicación dentro de la red social, con los atributos esenciales como el ID, contenido, usuario, fecha, entre otros.

- **Atributos:**

- id: Identificador único de la publicación.



Publicacion	
likeCount	int
content	String
usuariold	int
usuario	Usuario
username	String
likedByCurrentUser	boolean
comentarios	List<Comentario>
id	int
date	Date
likedBy	List<String>
formattedDate	String
commentCount	int

- username: Nombre de usuario del creador de la publicación.

- content: Contenido de la publicación.

- date: Fecha en que se realizó la publicación.

- likeCount: Número de "me gusta" que tiene la publicación.

- likedBy: Lista de usuarios que han dado "me gusta".

- likedByCurrentUser: Indica si el usuario actual le ha dado "me gusta".

- commentCount: Número de comentarios en la publicación.

- comentarios: Lista de comentarios asociados a la publicación.

- usuariold: ID del usuario asociado.

- usuario: Instancia del objeto Usuario asociada a la publicación.

- formattedDate: Fecha formateada en cadena de texto.

- **Constructores:**


- Un constructor vacío y otro que recibe parámetros básicos como el ID, username, contenido, fecha, y otros.

- **Métodos:**

- Métodos getter y setter para acceder y modificar los valores de los atributos.

PublicacionDAO.java

Esta clase es responsable de las operaciones de base de datos relacionadas con la entidad Publicacion. Se conecta a la base de datos utilizando JDBC para insertar y recuperar publicaciones.



PublicacionDAO	
obtenerPublicaciones()	List<Publicacion>
insertarPublicacion(Publicacion)	boolean

- **Atributos:**

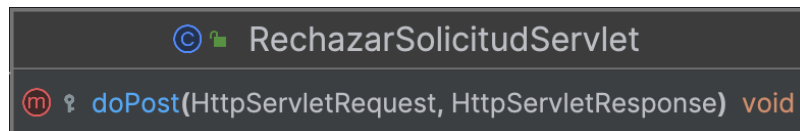
- DB_URL: URL de conexión a la base de datos (usando Derby en este caso).

- **Métodos:**

- insertarPublicacion: Inserta una nueva publicación en la base de datos.
- obtenerPublicaciones: Recupera todas las publicaciones de la base de datos, incluyendo información del usuario que las creó.

RechazarSolicitudServlet.java

Este servlet maneja la lógica para rechazar una solicitud de amistad en la red social.

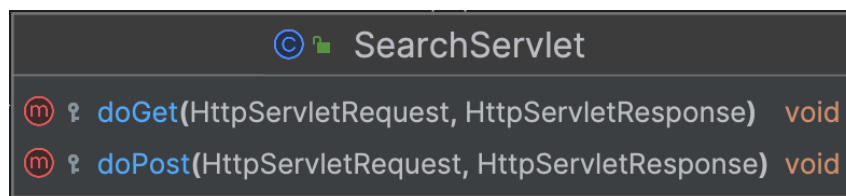


- **Método doPost:**

- Obtiene el ID de la solicitud a rechazar desde los parámetros de la petición.
- Utiliza el DAO de solicitudes de amistad (`FriendRequestDAO`) para ejecutar la acción de rechazar la solicitud.
- Redirige a la página de perfil o muestra un error si algo falla.

SearchServlet.java

Este servlet permite buscar usuarios y enviar solicitudes de amistad.



- **Atributos:**

- `usuarioDAO` y `friendRequestDAO`: Instancias de los DAOs para interactuar con la base de datos.

- **Métodos:**

- `doGet`: Verifica si hay una sesión activa y redirige a la página de búsqueda.
- `doPost`: Maneja las solicitudes de amistad, buscando usuarios y enviando solicitudes según los parámetros recibidos. Además, se encarga de gestionar la lógica de relaciones entre los usuarios, determinando si son amigos, tienen solicitudes pendientes, o están disponibles.

SolicitudAmistad.java

©	SolicitudAmistad	
Ⓜ	toString()	String
Ⓟ	emisorId	int
Ⓟ	id	int
Ⓟ	fecha	Date
Ⓟ	receptorId	int
Ⓟ	estado	String

- Representa una solicitud de amistad entre usuarios, con atributos como id, emisorId, receptorId, estado y fecha.
- El constructor vacío y el constructor con parámetros permiten crear objetos de esta clase de diversas formas.
- Los métodos getters y setters permiten manipular estos atributos de forma controlada. Además, se incluye un toString() para facilitar la depuración.

TimelineServlet.java

©	TimelineServlet	
Ⓜ	doGet(HttpServletRequest, HttpServletResponse)	void
Ⓜ	doPost(HttpServletRequest, HttpServletResponse)	void

- Se encarga de manejar tanto las solicitudes GET como POST relacionadas con el timeline del usuario.
- En doGet(), se obtienen todas las publicaciones de la base de datos, se calculan los comentarios y "likes" asociados a cada publicación, y se redirige a un JSP (timeline.jsp).
- En doPost(), gestiona la creación de nuevas publicaciones, comentarios y "likes". Si se recibe un "like" o un comentario, la acción correspondiente se maneja a través de la base de datos.

Usuario.java

- Representa a un usuario dentro del sistema, con atributos como id, fullName, email, username, password, profilePic, y listas para amigos y publicaciones.
- Los métodos getters y setters permiten manipular los atributos, y se incluye la gestión de relaciones de amistad y la visualización de publicaciones.

©	Usuario	
Ⓟ	password	String
Ⓟ	email	String
Ⓟ	isFriend	boolean
Ⓟ	currentUser	boolean
Ⓟ	fullName	String
Ⓟ	username	String
Ⓟ	publicaciones	List<Publicacion>
Ⓟ	friends	List<Usuario>
Ⓟ	relacionAmistad	String
Ⓟ	id	int
Ⓟ	profilePic	String

UsuarioDAO.java

©	UsuarioDAO	
Ⓜ	registrar(Usuario)	boolean
Ⓜ	obtenerUsuarioPorUsername(String)	Usuario
Ⓜ	login(String, String)	Usuario
Ⓜ	eliminarAmigo(int, int)	void
Ⓜ	tieneSolicitudPendiente(int, int)	boolean
Ⓜ	actualizarUsuario(Usuario)	boolean
Ⓜ	obtenerAmigos(int)	List<Usuario>
Ⓜ	agregarAmigo(String, String)	void
Ⓜ	esAmigo(int, int)	boolean
Ⓜ	obtenerPublicacionesPorUsuario(int)	List<Publicacion>
Ⓜ	buscarUsuarios(String, int)	List<Usuario>
Ⓜ	obtenerUsuarioPorId(int)	Usuario

- Esta clase maneja las operaciones de base de datos relacionadas con los usuarios, como el registro, inicio de sesión, obtención de amigos y la búsqueda de usuarios.
- En el método registrar(), se insertan nuevos usuarios en la base de datos. El método login() valida las credenciales de un usuario y recupera sus datos. Otros métodos gestionan la relación de amistad, como obtenerAmigos() y agregarAmigo().

Creación de tablas para la Base de Datos

En este apartado, se describe el proceso de diseño e implementación de las tablas necesarias para la base de datos del sistema.

A continuación, se presenta el código SQL utilizado para la creación de dichas tablas, seguido de una evidencia visual que confirma su correcta implementación en el gestor de base de datos seleccionado.

Código SQL:

```
CREATE TABLE Usuario (  
  id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,  
  fullName VARCHAR(255) NOT NULL,  
  email VARCHAR(255) NOT NULL,  
  username VARCHAR(100) NOT NULL UNIQUE,  
  password VARCHAR(255) NOT NULL,  
  profilepic VARCHAR(255)  
);
```

```
CREATE TABLE PUBLICACION (  
  
  ID INT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,  
  
  USUARIO_ID INT NOT NULL,  
  
  USERNAME VARCHAR(255) NOT NULL,  
  
  CONTENT VARCHAR(2000) NOT NULL,  
  
  DATE TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,  
  
  IMAGE_URL VARCHAR(500),  
  
  LIKE_COUNT INT DEFAULT 0,  
  
  LIKED_BY CLOB,  
  
  COMMENT_COUNT INT DEFAULT 0,  
  
  COMMENTS CLOB,  
  
  FOREIGN KEY (USUARIO_ID) REFERENCES USUARIO(ID)  
  
);
```

```
CREATE TABLE LIKES (  
    ID INT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,  
    PUBLICACION_ID INT,  
    USUARIO_ID INT,  
    FOREIGN KEY (PUBLICACION_ID) REFERENCES PUBLICACION(ID),  
    FOREIGN KEY (USUARIO_ID) REFERENCES USUARIO(ID)  
);
```

```
CREATE TABLE COMENTARIOS (  
    ID INT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,  
    PUBLICACION_ID INT,  
    USUARIO_ID INT,  
    CONTENIDO VARCHAR(255),  
    FECHA TIMESTAMP,  
    FOREIGN KEY (PUBLICACION_ID) REFERENCES PUBLICACION(ID),  
    FOREIGN KEY (USUARIO_ID) REFERENCES USUARIO(ID)  
);
```

```
CREATE TABLE SOLICITUDES_AMISTAD (  
    ID INT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,  
    EMISOR_ID INT NOT NULL,  
    RECEPTOR_ID INT NOT NULL,  
    ESTADO VARCHAR(20) DEFAULT 'pendiente',    FECHA TIMESTAMP  
    DEFAULT CURRENT_TIMESTAMP,  
    FOREIGN KEY (EMISOR_ID) REFERENCES USUARIO(ID),  
    FOREIGN KEY (RECEPTOR_ID) REFERENCES USUARIO(ID)  
);
```

```
CREATE TABLE AMIGOS (  
    USER1 INT NOT NULL,  
    USER2 INT NOT NULL,  
    PRIMARY KEY (USER1, USER2),  
    FOREIGN KEY (USER1) REFERENCES USUARIO(ID),  
    FOREIGN KEY (USER2) REFERENCES USUARIO(ID)  
);
```

```
CREATE TABLE MENSAJES (  
    ID INT PRIMARY KEY GENERATED ALWAYS AS IDENTITY (START WITH  
    1, INCREMENT BY 1),  
    EMISOR_ID INT NOT NULL,  
    RECEPTOR_ID INT NOT NULL,  
    CONTENIDO VARCHAR(255) NOT NULL,  
    FECHA TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
    FOREIGN KEY (EMISOR_ID) REFERENCES USUARIO(ID),  
    FOREIGN KEY (RECEPTOR_ID) REFERENCES USUARIO(ID)  
);
```

Inserción de datos de prueba

```
INSERT INTO USUARIO (fullName, email, username, password, profilepic)
VALUES
('Juan Pérez', 'juanperez@example.com', 'juanperez', 'password123',
'images/default-profile-pic.jpeg'),
('Ana Gómez', 'anagomez@example.com', 'anagomez', 'password456',
'images/default-profile-pic.jpeg'),
('Carlos López', 'carloslopez@example.com', 'carloslopez', 'password789',
'images/default-profile-pic.jpeg'),
('Daniel Senespleda', 'daniel.senespleda@gmail.com', 'dani.sf', 'Dani',
'dani.jpeg'),
('Alejandro Dario Albert', 'alejandro.dario@gmail.com', 'alexalbert12', 'Dario',
'dario.jpg');
```

```
INSERT INTO PUBLICACION (USUARIO_ID, USERNAME, CONTENT,
IMAGE_URL, LIKE_COUNT, LIKED_BY, COMMENT_COUNT, COMMENTS)
VALUES
(1, 'juanperez', 'Este es el contenido de mi primera publicación',
'http://image1.com', 10, ['ana_gomez', 'carlos_lopez'], 2, ["Comentario 1",
"Comentario 2"]),
(2, 'anagomez', 'Publicación de prueba de Ana', 'http://image2.com', 5,
['juanperez'], 1, ["Comentario 1"]),
(3, 'carloslopez', 'Mi primer post como Carlos', 'http://image3.com', 0, [], 0, []);
```

```
INSERT INTO SOLICITUDES_AMISTAD (EMISOR_ID, RECEPTOR_ID,
ESTADO)
VALUES
(1, 2, 'pendiente'),
(2, 3, 'aceptada'),
(3, 1, 'rechazada');
```

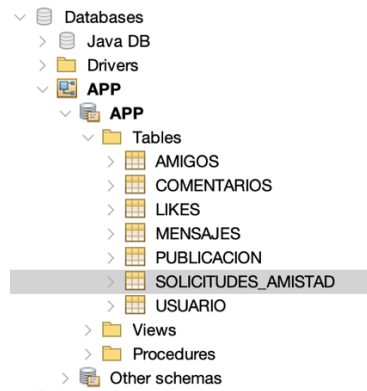
```
INSERT INTO AMIGOS (USER1, USER2)
VALUES
(2, 3),
(1, 2);
```

```
INSERT INTO MENSAJES (EMISOR_ID, RECEPTOR_ID, CONTENIDO)
VALUES
(1, 2, 'Hola Ana, ¿cómo estás?'),
(2, 1, '¡Hola Juan! Estoy bien, gracias por preguntar.'),
(3, 1, 'Carlos aquí, ¿todo bien?');
```

```
INSERT INTO COMENTARIOS (PUBLICACION_ID, USUARIO_ID,
CONTENIDO, FECHA)
VALUES
(1, 2, 'Muy buena publicación, Juan!', CURRENT_TIMESTAMP),
(1, 3, '¡Totalmente de acuerdo!', CURRENT_TIMESTAMP),
(2, 1, 'Me encanta lo que escribes, Ana', CURRENT_TIMESTAMP);
```

```
INSERT INTO LIKES (PUBLICACION_ID, USUARIO_ID)
VALUES
(1, 2),
(1, 3),
(2, 1);
```

Base de Datos



Patrón MVC

Modelo (Model)

El modelo representa la lógica de negocio y la interacción con la base de datos. Incluye las clases que representan los datos y las operaciones que se pueden realizar sobre ellos.

Ejemplo de clases del modelo:

Usuario.java
Publicacion.java
Comentario.java
SolicitudAmistad.java
Mensaje.java
Ejemplo de clases DAO:

UsuarioDAO.java
PublicacionDAO.java
ComentarioDAO.java
FriendRequestDAO.java
MessageDAO.java
LikeDAO.java

Vista (View)

La vista es responsable de la presentación de los datos. En este caso, utilizamos JSP para generar el HTML que se enviará al cliente.

Ejemplo de archivos JSP:

login.jsp
register.jsp
timeline.jsp
profile.jsp
search.jsp
messages.jsp
admin.jsp

Controlador (Controller)

El controlador maneja las solicitudes del usuario, interactúa con el modelo para obtener los datos necesarios y selecciona la vista adecuada para mostrar esos datos.

Ejemplo de Servlets:

AuthServlet.java
TimelineServlet.java
ProfileServlet.java
SearchServlet.java
MessagesServlet.java
AdminServlet.java
AceptarSolicitudServlet.java
RechazarSolicitudServlet.java

- Modelo: Clases Java que representan los datos y la lógica de negocio.
 - Vista: Archivos JSP que generan el HTML para la presentación.
 - Controlador: Servlets que manejan las solicitudes del usuario, interactúan con el modelo y seleccionan la vista adecuada.
- Esta estructura sigue el patrón MVC, separando claramente la lógica de negocio, la presentación y el control de flujo de la aplicación.