Análisis de meneáme - GitHub

Se ha llevado a cabo una revisión completa de la estructura y el funcionamiento interno de Menéame.net, desgranando sus módulos principales y la manera en que cada uno interactúa con la base de datos. A lo largo del análisis, se han documentado los flujos fundamentales: autenticación de usuarios, publicación de enlaces, gestión de comentarios, votaciones (karma), moderación y administración, así como la integración con scripts automáticos y la API. Del mismo modo, se ha descrito la disposición de las tablas y los modelos clave, sentando la base para reproducir la lógica de Menéame en un framework moderno. Según Benjamí, actualmente meneáme es en un 85% este código analizado. Faltan otros servicios que son directos como nótame, fisgona, etc.

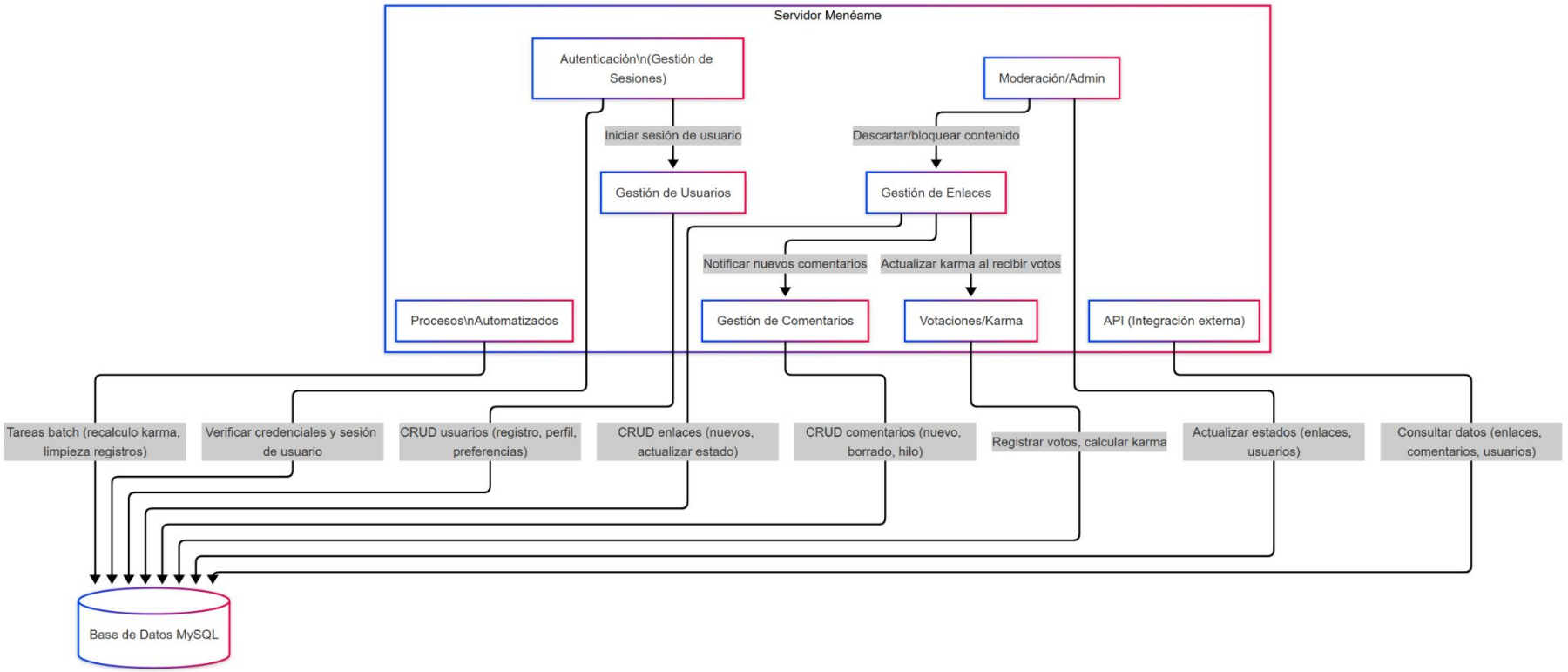
Objetivo y Beneficios

Este trabajo apunta a desacoplar la lógica de negocio de las tareas de integración y orquestación de datos, ganando en escalabilidad, visibilidad y velocidad para la creación o modificación de flujos. Esta separación resulta especialmente valiosa cuando se manejan múltiples fuentes de datos o grandes volúmenes de información de forma asíncrona.

Diagrama de Componentes

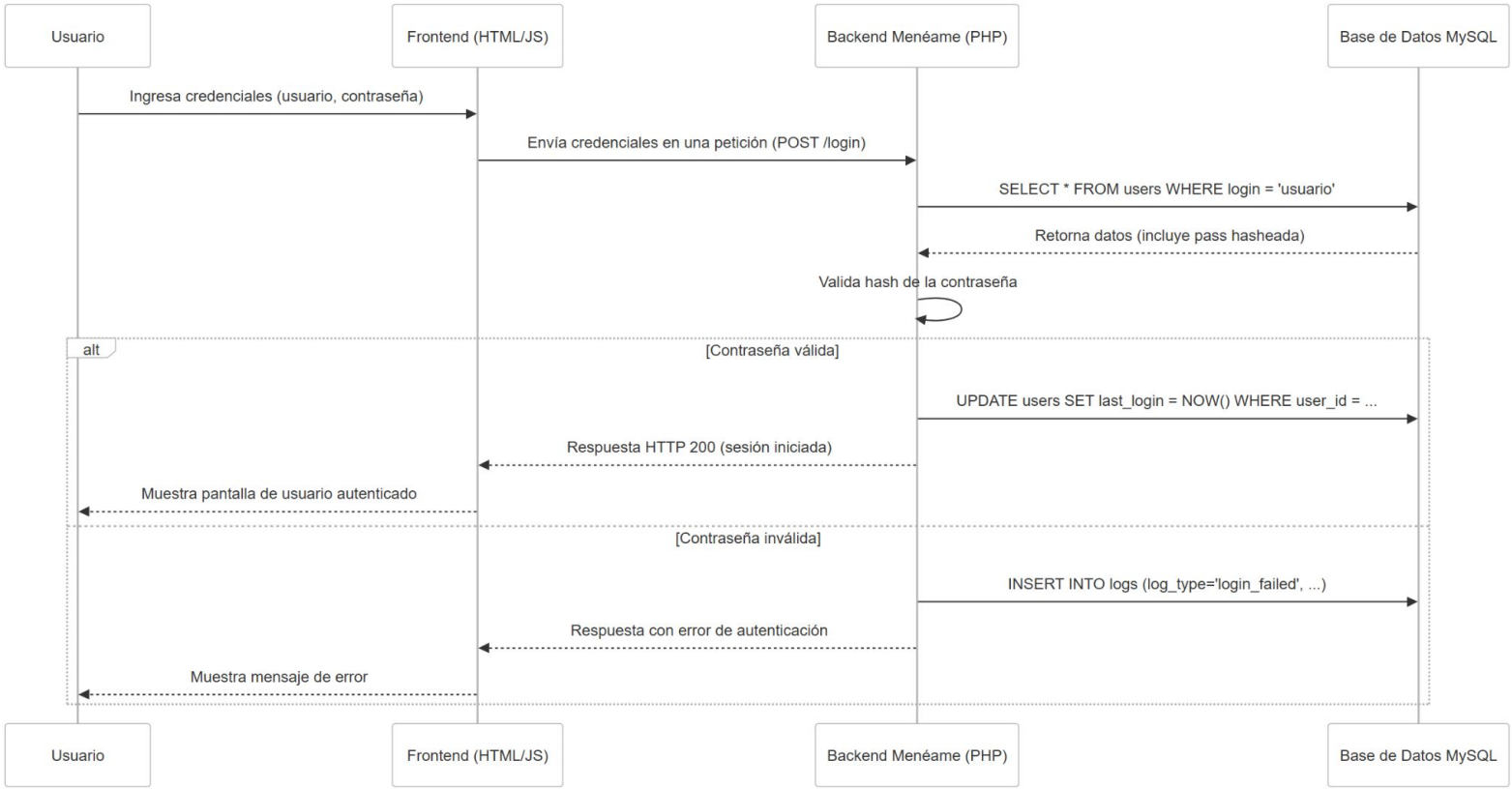
En la arquitectura de Menéame se identifican múltiples módulos o componentes principales y sus interacciones. Los componentes clave incluyen:

- **Autenticación/Usuarios**,
- **Enlaces/Noticias**,
- **Votaciones/Karma**,
- **Comentarios**,
- **Moderación/Administración**,
- **API** y **Procesos Automáticos**.



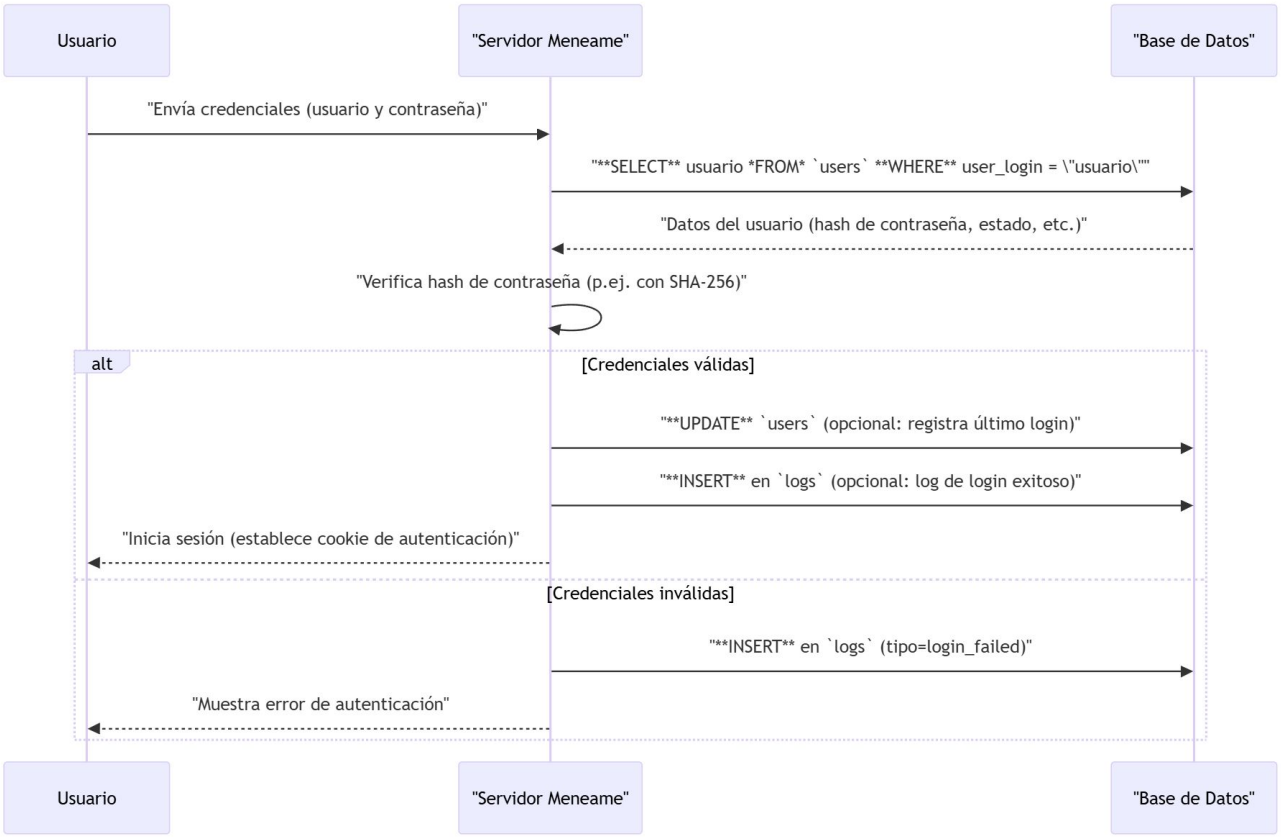
Diagramas de Secuencia

Diagramas de Secuencia: A continuación se presentan diagramas de secuencia UML para los principales flujos del sistema, incluyendo todas las llamadas a base de datos relevantes. Estos diagramas ilustran paso a paso la interacción entre actores (usuarios o administradores), el sistema Menéame (backend) y la base de datos, para lograr cada funcionalidad. Cada secuencia muestra cómo circula la información y en qué puntos se realizan operaciones en la base de datos, lo cual será muy útil para replicar el comportamiento en otro entorno, como en este ejemplo:



Autenticación y Gestión de Usuarios (Login, Registro, Perfiles)

a. Secuencia de *Login* (inicio de sesión): El usuario ingresa sus credenciales y el sistema valida la información contra la base de datos de usuarios. Si son correctas, inicia la sesión; si no, devuelve un error.



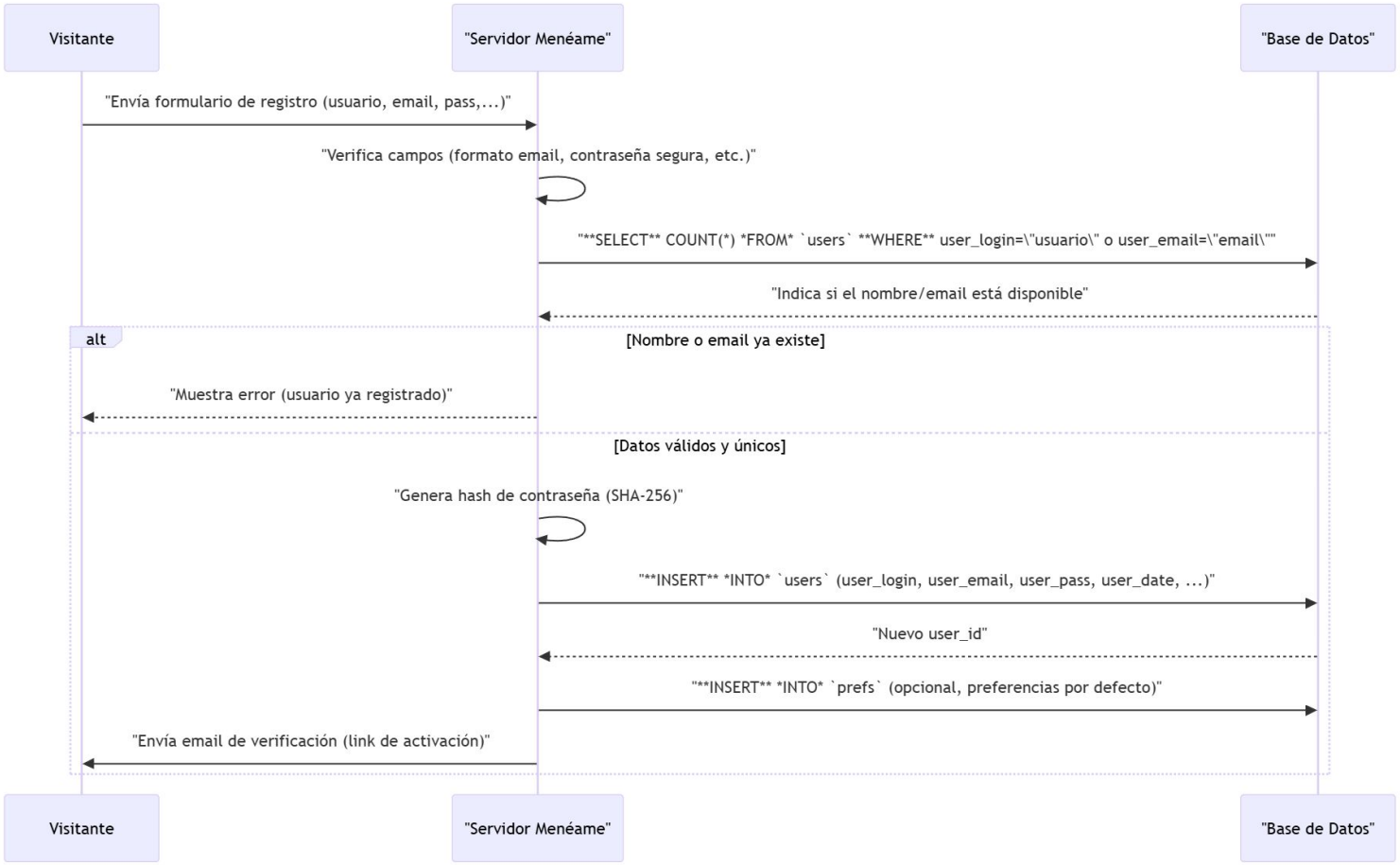
Autenticación y Gestión de Usuarios (Login, Registro, Perfiles)

Secuencia de *Login* (inicio de sesión)
Descripción: En esta secuencia, el **Usuario** envía sus credenciales al servidor.

1. El **Servidor Menéame** (backend) consulta la **Base de Datos** (`users`) para obtener el registro del usuario (incluyendo el hash de la contraseña almacenado).
2. Luego el servidor compara la contraseña proporcionada con el hash guardado usando el algoritmo de hash definido (por ejemplo SHA-256).
3. Si coinciden y el usuario no está marcado como deshabilitado, se considera una autenticación exitosa.
4. El sistema entonces podría actualizar algunos datos del usuario (último acceso) y registra la sesión activa (por medio de cookies en el navegador).
5. Finalmente responde al usuario indicando que la sesión fue iniciada (normalmente mediante un **cookie** de sesión `u` y `ukey` que se guardan en el navegador para autenticar futuras peticiones).
6. En caso de credenciales incorrectas, el servidor registra el intento fallido (p. ej. inserta un registro en la tabla de *logs* con `log_type = "login_failed"`) y notifica al usuario que la autenticación falló.
7. Tras un login exitoso, el objeto de usuario autenticado queda disponible en sesión; muchas páginas incluirán la lógica de comprobación de sesión (usando el módulo de **Autenticación**) para determinar si el usuario está o no logueado en cada petición (esto se maneja mediante cookies y la clase `UserAuth` que valida `$_COOKIE['u']` y `$_COOKIE['ukey']` en cada request).

Autenticación y Gestión de Usuarios (Login, Registro, Perfiles)

b. Secuencia de *Registro* de nuevo usuario: El visitante llena el formulario de registro, el sistema valida y crea un nuevo usuario en la base de datos, enviando un correo de confirmación.



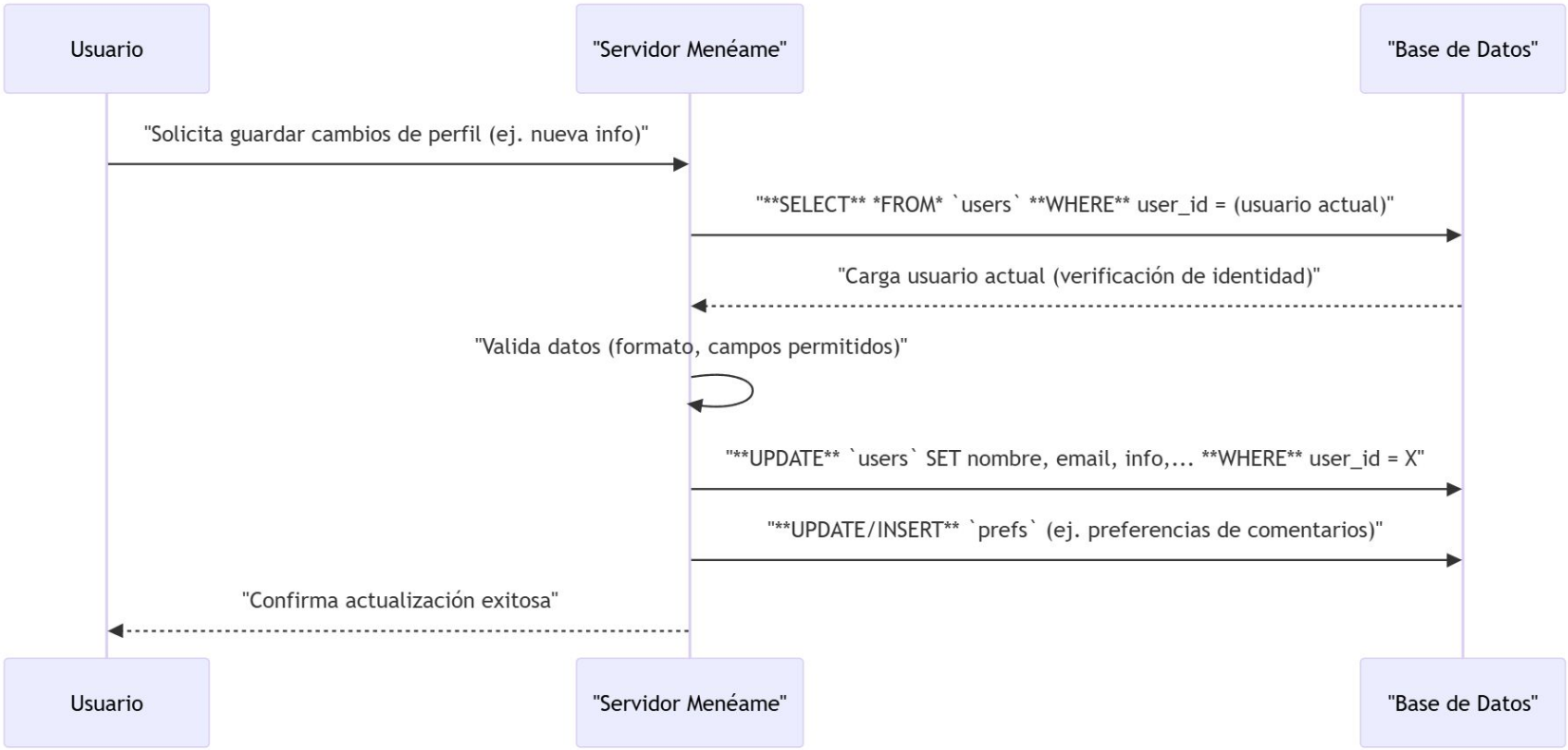
Autenticación y Gestión de Usuarios (Login, Registro, Perfiles)

Secuencia de *Registro* de nuevo usuario
Descripción: En esta secuencia de **Registro**, un **Visitante** (no autenticado) envía sus datos de registro.

1. El **Servidor** valida localmente el formato de los datos (por ejemplo, que el email tenga un formato correcto, que la contraseña cumpla requisitos mínimos y que se haya superado un captcha de seguridad).
2. Luego comprueba en la **BD** si ya existe un usuario con el mismo nombre de usuario o correo electrónico (para evitar duplicados).
3. Si el nombre o email ya están en uso, se informa al visitante para que elija otros datos. Si todo es válido, el servidor procede a registrar el nuevo usuario: genera el *hash* de la contraseña (Menéame utiliza SHA-256 con *salt* aleatorio para almacenar contraseñas), y realiza un **INSERT** en la tabla `users` con los datos proporcionados (nombre de usuario, email, hash de contraseña, fecha de registro, IP, etc.). Inicialmente, el nuevo usuario suele tener `user_level = "normal"` y un campo `user_validated_date` nulo (indicando que aún no se ha validado por correo). El sistema podría también crear registros asociados, por ejemplo entradas por defecto en la tabla de preferencias (`prefs`) para ese usuario (como la preferencia de subscripción por defecto).
4. Tras crear al usuario en la base, el servidor envía un **correo de verificación** a la dirección registrada, con un enlace único. El flujo de registro concluye mostrando al usuario un mensaje para que revise su email.
5. **Paso de activación:** Cuando el usuario hace clic en el enlace de verificación (por ejemplo, URL con un token único), el servidor recibe esa petición, busca al usuario correspondiente (según el token o ID codificado), y marca el usuario como validado – típicamente haciendo un **UPDATE** `users` seteando `user_validated_date = NOW()` y quizás ajustando el nivel (de un estado provisional a activo). Solo entonces el usuario puede iniciar sesión.
6. (El código de Menéame efectivamente borra automáticamente usuarios que no se validaron tras cierto tiempo, por lo que este paso es crucial). Después de la validación, el usuario ya puede loguearse normalmente.

Autenticación y Gestión de Usuarios (Login, Registro, Perfiles)

b. Secuencia de *Actualización de Perfil*: Un usuario autenticado puede modificar su perfil (p. ej. cambiar contraseña, correo alternativo, avatar, información pública). El flujo siguiente ilustra la edición de perfil, centrado en los accesos a base de datos:



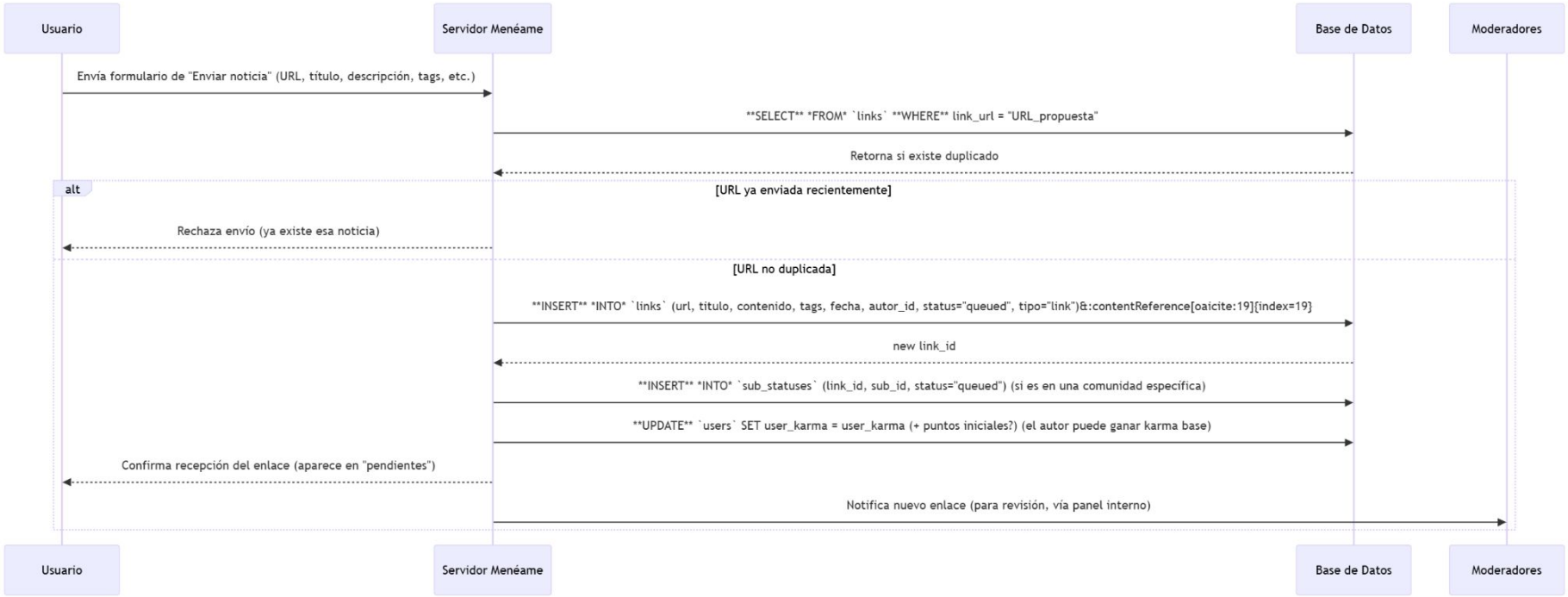
Autenticación y Gestión de Usuarios (Login, Registro, Perfiles)

Secuencia de *Actualización de Perfil*
Descripción: Para la **Actualización de Perfil**, el **Usuario** envía los cambios deseados (por ejemplo, cambiar su descripción pública, enlace personal, avatar, ajustes de notificaciones, etc.).

1. El **Servidor** primero podría volver a cargar al usuario desde la base de datos para asegurarse de que el usuario en sesión es válido y obtener datos actuales.
2. Luego valida que los nuevos valores cumplan los requisitos (por ejemplo, tamaño de avatar, formato de URL).
3. A continuación realiza las operaciones necesarias en la **BD**: típicamente un **UPDATE** en la tabla `users` para los campos del perfil que se han modificado (nombre público, URL personal, etc.), y también **INSERT/UPDATE** en la tabla de `prefs` si se cambian preferencias almacenadas separadamente (por ejemplo, orden de comentarios preferido, subscripción por defecto a ciertas categorías, etc.). En el código de Menéame, por ejemplo, `user_public_info`, `user_url`, `user_avatar` se almacenan en la tabla principal de usuarios, mientras que algunas preferencias específicas se guardan en `prefs` (clave-valor).
4. Tras actualizar la base de datos, el sistema notifica al usuario que su perfil ha sido guardado correctamente.
5. Este flujo de perfil incluye también el caso de cambio de contraseña, donde se calcularía un nuevo hash y se actualizaría `user_pass` (posiblemente invalidando sesiones activas).
6. Todos estos pasos aseguran que la información del usuario esté sincronizada en la base de datos

Publicación y Gestión de Enlaces/Noticias

a. Secuencia de *Envío de nueva noticia*: Un usuario registrado envía un enlace o noticia. El sistema guarda la propuesta en la base de datos con estado "pendiente" (cola de nuevas) y realiza comprobaciones adicionales.



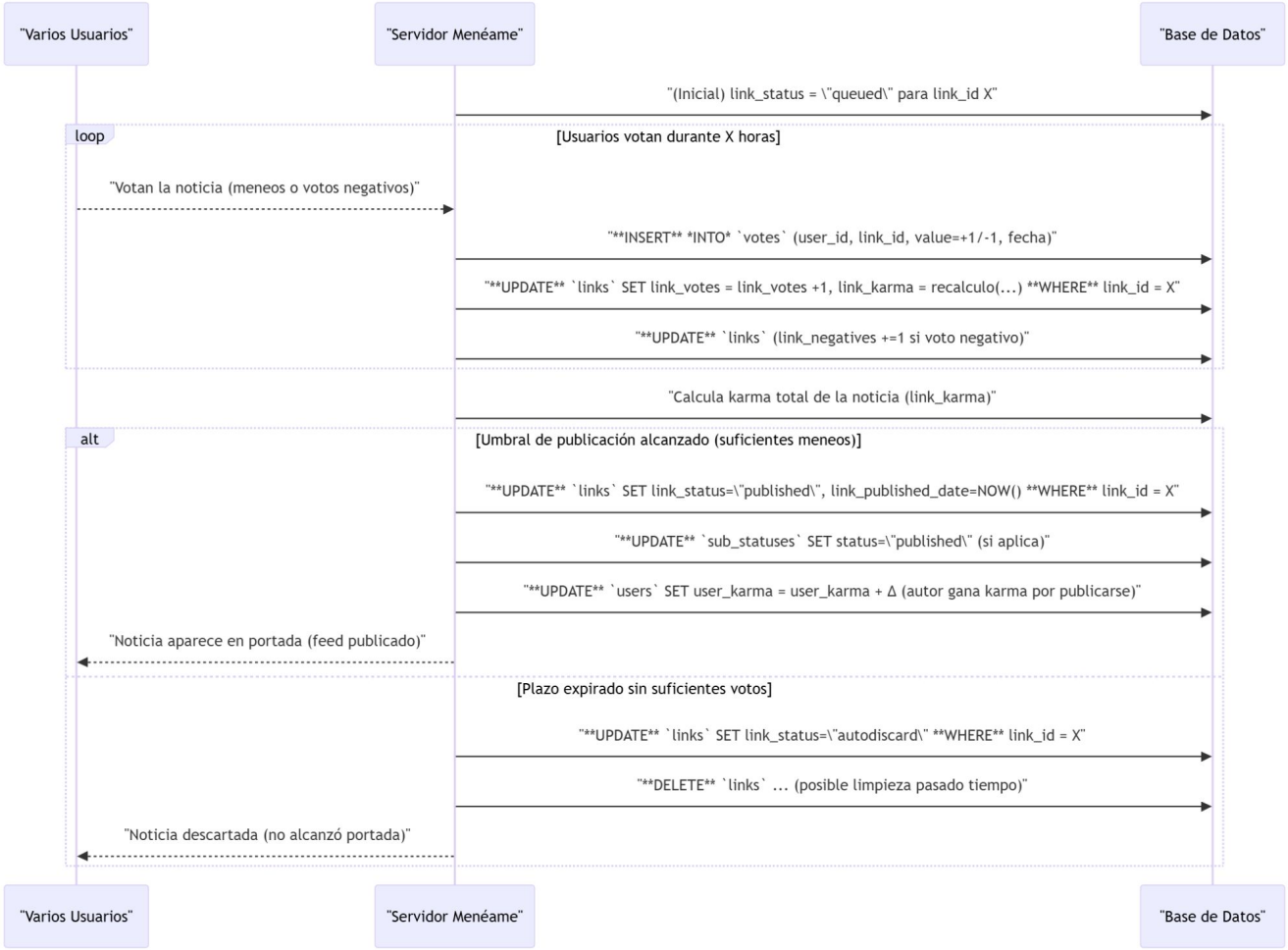
Publicación y Gestión de Enlaces/Noticias

Secuencia de *Envío de nueva noticia*
Descripción: En el flujo de **Enviar enlace**, el **Usuario** completa un formulario con los datos de la noticia: la URL (enlaces externos) o texto (si es un artículo propio), título, descripción/resumen, etiquetas (*tags*) y posiblemente la comunidad (*sub*) donde desea enviarla.

1. Al recibir estos datos, el **Servidor** primero comprueba si la URL ya fue enviada recientemente para evitar duplicados: realiza una consulta SELECT en la tabla `links` buscando alguna entrada con el mismo `link_url` todavía activa.
2. Si encuentra una noticia duplicada (especialmente si está en pendientes o publicada), puede abortar el proceso devolviendo un mensaje de error al usuario (invitando quizás a votar la existente).
3. Si no hay duplicado, el servidor procede a **insertar** la nueva noticia en la base de datos. En la tabla `links`, se crea un nuevo registro con campos como el *título*, *URL*, *contenido* (cuerpo o resumen), *tags*, la fecha/hora actual (`link_date`), el identificador del autor (`link_author`), y un estado inicial `link_status = "queued"` (pendiente). También se registra el tipo de contenido (`link_content_type`), por ejemplo `"link"` para noticias con URL o `"article"` para contenido original escrito, ya que Menéame distingue ambos casos. El nuevo registro tendrá contadores de votos y comentarios inicialmente a 0 (p. ej. `link_votes=0`, `link_negatives=0`, `link_comments=0`, `link_karma=0`).
4. Si el envío se realizó a una **sub** (comunidad temática) en lugar del Menéame general, se registra adicionalmente en la tabla `sub_statuses` una relación que indica que el `link_id` pertenece a cierto `sub_id` con estado "queued". (El sistema de subs permite que una noticia esté pendiente en una comunidad; `sub_statuses` también se usa para seguir el estado y origen de enlaces en comunidades).
5. Tras insertar, el servidor podría actualizar ciertas cosas: por ejemplo, podría recalcular el karma del usuario autor sumando un pequeño karma inicial por la aportación (aunque generalmente el karma de usuario se ajusta más tarde cuando la noticia se publica o es votada; en Menéame suele haber un karma base y luego puntos por publicaciones exitosas).
6. En este punto, el usuario recibe una confirmación de que su noticia fue enviada con éxito, y la verá listada en la sección de *pendientes/nuevas*.
7. Internamente, el staff de moderación o un proceso automático podría ser notificado de la nueva entrada (esto puede ser vía una cola interna, o simplemente observando la lista de pendientes).

Publicación y Gestión de Enlaces/Noticias

b. Secuencia de *Promoción a portada o descarte (gestión de la noticia)*: Tras el envío, la noticia permanece en **cola de nuevas** y es sometida a votación por la comunidad. Este flujo ilustra cómo una noticia pasa de *pendiente* a *publicada* en portada al alcanzar cierto umbral de votos (karma), o bien es descartada automáticamente por falta de apoyo o manualmente por un administrador.



Publicación y Gestión de Enlaces/Noticias

Secuencia de *Promoción a portada o descarte (gestión de la noticia)*
Descripción: Aquí vemos el **ciclo de vida** de una noticia en pendientes. Una vez en la cola, múltiples **Usuarios votantes** pueden interactuar con ella: **votar positivo** (“meneo”) para promocionarla, o **votar negativo** (por diversas razones, como *spam* o irrelevante).

1. Cada voto genera una transacción similar a la descrita en el sistema de votaciones (ver siguiente sección): se inserta un registro en la tabla `votes` indicando qué usuario votó qué contenido y con qué valor, y se actualiza el registro de `links` correspondiente incrementando el contador de votos (`link_votes`) o de negativos (`link_negatives`) según corresponda.
2. También se recalcula el *karma* acumulado de la noticia (`link_karma`), que Menéame calcula en función de los votos positivos menos ciertos multiplicadores de negativos, entre otros factores.
3. Durante un período determinado (p. ej. unas pocas horas), la noticia acumula votos. El backend periódicamente (o tras cada voto) evalúa si se ha superado el **umbral de publicación**: este umbral es una combinación de número de votos, relación de positivos/negativos y tiempo en cola.
4. Si la noticia alcanza suficientes **meneos** (votos positivos netos) antes de expirar el tiempo, entonces el sistema la promueve a la portada. Esto se realiza cambiando su estado: un **UPDATE** en la tabla `links` marca `link_status = "published"` y registra la hora de publicación (`link_published_date`).
5. Si la noticia estaba asociada a una *sub*, su entrada en `sub_statuses` también se actualiza a "published".
6. Al publicarse, típicamente el autor recibe un incremento en su karma de usuario (según las reglas, p. ej. +% determinado que el script de karma consolidará) y los usuarios que votaron reciben eventualmente algún crédito.
7. La noticia ahora es visible para todos en la portada (el servidor la incluirá en las consultas de noticias publicadas).
8. En caso contrario, si transcurre el tiempo límite y no logra los votos necesarios, la noticia es **descartada automáticamente** – se actualiza `link_status` a "autodiscard". Es posible que un proceso de limpieza elimine noticias autodiscard después de unos días para no llenar la base (como se ve en el script de karma se borran enlaces descartados viejos que cumplan ciertas condiciones).
9. Los usuarios notarán que la noticia desaparece de la lista de pendientes y no llega a portada. (Un administrador también podría descartar la noticia manualmente antes de tiempo; eso se ve en Moderación).
10. En resumen, este diagrama cubre la gestión automática principal: publicación o descarte según votos.

Sistema de Votaciones (Karma, votos a noticias y comentarios)

a. Secuencia de *Voto a una noticia:* Un usuario vota positiva o negativamente una noticia (en cola o publicada). El sistema registra el voto y actualiza los conteos y karma correspondientes.

Sistema de Votaciones (Karma, votos a noticias y comentarios)

Secuencia de *Voto a una noticia*

Descripción:

1. Cuando un **Usuario** realiza un voto sobre una **noticia**, el **Servidor** primero puede comprobar si ese usuario ya votó anteriormente esa misma noticia (para impedir votos duplicados); esto se haría consultando la tabla `votes` filtrando por `vote_user_id` y `vote_link_id`.
2. Si ya existe un registro (el usuario ya votó), el sistema podría bloquear el nuevo voto o en algunos casos permitir cambiar el sentido (Menéame tradicionalmente no permite votar dos veces la misma noticia). Asumiendo que es un voto nuevo válido, el servidor inserta un registro en la tabla `votes` indicando el tipo de contenido = "links", el identificador de la noticia, el usuario que vota y el valor (+1 para voto positivo, -1 para negativo).
3. Luego actualiza los campos agregados en la tabla de `links`: incrementa `link_votes` si es positivo, o `link_negatives` si es negativo, y recalcula el `link_karma`. El karma de la noticia típicamente se calcula como la suma ponderada de votos; en el código de Menéame se utilizan fórmulas que restan un múltiplo de los negativos, etc., para determinar la relevancia.
4. Tras el update, la base devuelve los nuevos totales y el servidor puede reflejar inmediatamente en la interfaz el nuevo contador de votos.
5. Adicionalmente, si la noticia estaba pendiente (`queued`), aquí o en un paso aparte se evalúa si con este voto se alcanzó el umbral de publicación; de ser así, se cambiará su estado a **"published"** y se fijará la hora de publicación (este paso lo mostramos en el diagrama anterior).
6. El efecto del voto en el **karma de usuario** no es inmediato en un solo voto – en Menéame, el karma del usuario votante puede verse afectado sutilmente si vota muchas noticias que acaban descartadas o si emite muchos votos negativos sin justificación (el sistema de karma penaliza ciertos patrones). Estas penalizaciones o recompensas se calculan en lotes (ver procesos automáticos).
7. Sin embargo, el **karma del autor de la noticia** sí se verá afectado al publicarse la noticia (ganando puntos) o ser descartada (no gana nada, o podría perder si hay abuso).

Sistema de Votaciones (Karma, votos a noticias y comentarios)

b. Secuencia de *Voto a un comentario:* Los usuarios también pueden votar comentarios individuales. El flujo es similar: se registra el voto y se actualiza la puntuación del comentario.

Sistema de Votaciones (Karma, votos a noticias y comentarios)

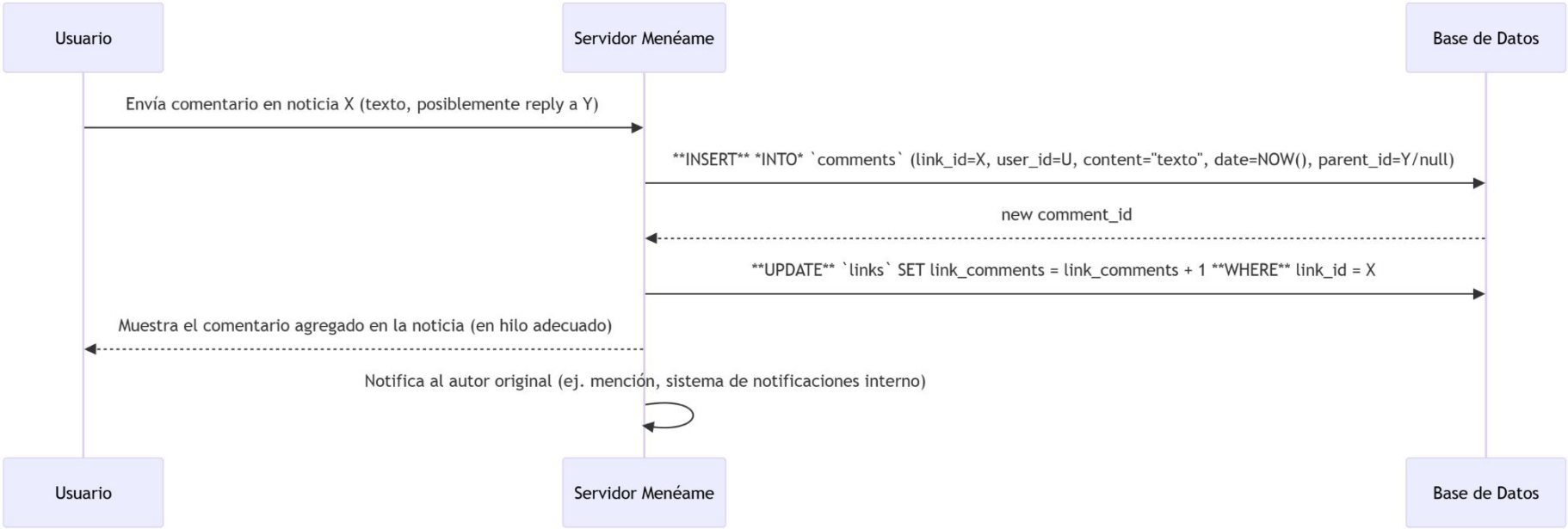
Secuencia de *Voto a un comentario*

Descripción: Un **comentario** puede recibir votos (en Menéame, los comentarios se pueden votar positivo o negativo una vez que el usuario tiene cierto karma). La secuencia es análoga a la de votar una noticia:

1. El **Servidor** verifica que el usuario no haya votado el comentario antes, luego inserta el voto en la tabla `votes` con `vote_type="comments"` y el ID del comentario.
2. Acto seguido, actualiza en la tabla `comments` los contadores de votos del comentario (Menéame podría no almacenar un contador explícito en `comments`, podría calcular el score sumando positivos y negativos desde `votes` al vuelo; pero para este modelo asumiremos que sí mantiene un campo de score o karma de comentario).
3. El **score del comentario** se recalcula (por ejemplo, #positivos - #negativos). Si un comentario acumula muchos negativos, el sistema podría marcarlo como *controvertido* u ocultarlo automáticamente (colapsado) para los usuarios por debajo de cierto karma, etc. Esto se representa con la condición final: si el puntaje cae por debajo de un umbral, el comentario podría cambiar de estado (p. ej. un campo `comment_status="hidden"` o simplemente una clase CSS cliente).
4. En cuanto al **karma de usuarios**, los votos a comentarios generalmente no afectan el karma del autor del comentario de forma acumulativa en Menéame (aunque un comentario muy votado puede dar trofeos o simplemente reflejarse en su reputación informal).
5. Sin embargo, los votos negativos excesivos de un usuario a comentarios podrían influir en su karma si abusa, similar al caso de noticias (esto también se gestiona en la rutina de karma).
6. En resumen, la inserción en `votes` y la actualización correspondiente del comentario son los pasos clave.

Gestión de Comentarios (añadir, eliminar, jerarquía de hilos)

a. Secuencia de *Añadir comentario:* Un usuario añade un nuevo comentario a una noticia (o en respuesta a otro comentario). El sistema guarda el comentario y actualiza la relación con la noticia.



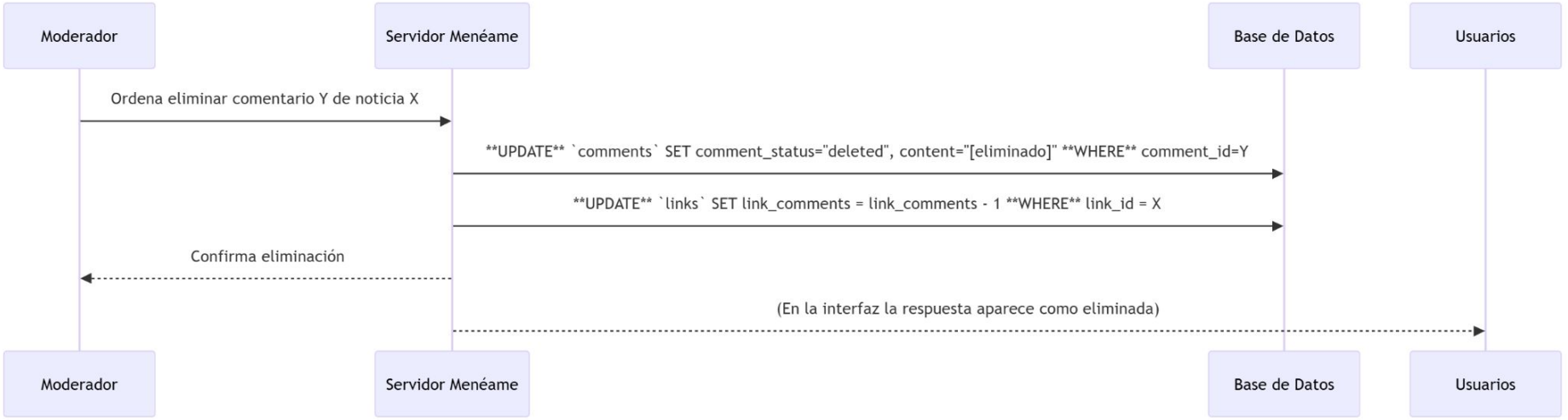
Gestión de Comentarios (añadir, eliminar, jerarquía de hilos)

Secuencia de *Añadir comentario*
Descripción: Para **publicar un comentario**, el **Usuario** envía el contenido de texto del comentario. Puede ser un comentario raíz a la noticia o una respuesta a otro comentario (en cuyo caso se incluye la referencia al comentario padre Y).

1. El **Servidor** inserta el nuevo comentario en la tabla `comments` con los datos: el identificador de la noticia (`comment_link_id`), el autor (`comment_user_id`), el texto (`comment_content`), la fecha/hora actual (`comment_date`), y si es respuesta a alguien, el `comment_parent_id` apuntando al comentario Y. Esta inserción devuelve un `comment_id` nuevo.
2. A continuación, se incrementa el contador de comentarios de la noticia correspondiente: se hace un **UPDATE** en `links` aumentando `link_comments` en 1. Esto permite llevar la cuenta de comentarios sin tener que contarlos cada vez.
3. Tras guardar, el servidor envía de vuelta al cliente la representación del comentario para que se visualice inmediatamente en la interfaz debajo de la noticia, en la posición adecuada de la jerarquía (por ejemplo, anidado bajo su comentario padre si lo tiene).
4. Menéame maneja la **jerarquía de hilos** a través del campo parent: cada comentario conoce su padre inmediato. Para mostrar todo el hilo, el sistema obtiene todos los comentarios de la noticia (por ejemplo con una consulta `SELECT * FROM comments WHERE link_id=X ORDER BY date` y luego los indenta según parent_id).
5. Alternativamente, puede usar un campo de *thread* o *path* para ordenar jerárquicamente, pero dado que en el código disponible se ve uso de parent solamente, la jerarquía se reconstruye recursivamente.
6. Finalmente, el sistema de Menéame podría notificar a ciertos usuarios del nuevo comentario: por ejemplo, si el comentario es respuesta a otro usuario, puede generar una **notificación interna** para ese usuario (esto en la implementación original se logra mediante la tabla `notifys` o similar, aunque no detallada aquí). También se registra el evento en *logs* (log de tipo `comment_new`) para auditoría.

Gestión de Comentarios (añadir, eliminar, jerarquía de hilos)

b. Secuencia de *Eliminación de comentario:* Un moderador (o el propio autor, bajo ciertas condiciones) decide eliminar un comentario. Se ilustra la acción de un administrador eliminando un comentario ofensivo.



Gestión de Comentarios (añadir, eliminar, jerarquía de hilos)

Secuencia de *Eliminación de comentario*
Descripción: Cuando se elimina un comentario, típicamente un **Moderador** inicia la acción (aunque Menéame permite al autor borrar su comentario en pocos minutos tras publicarlo, usualmente luego los comentarios solo pueden ser eliminados por administración) se genera este flujo:

1. El **Servidor** procesa la petición de borrado: en la base de datos, la acción más común es marcar el comentario como eliminado en lugar de borrarlo físicamente. En nuestro diagrama se muestra un **UPDATE** en la tabla `comments` cambiando un campo de estado (por ejemplo `comment_status`) a "deleted" y quizá sobrescribiendo el contenido con un mensaje tipo "[eliminado por el autor]" o "[comentario eliminado por la administración]". De esta forma se conserva el registro para mantener la estructura del hilo (las respuestas al comentario eliminado siguen existiendo, pero se indica que el padre fue eliminado).
2. También podría establecerse algún indicador de que ya no debe mostrarse el avatar/usuario del autor (dependiendo de políticas de la plataforma).
3. Adicionalmente, el sistema podría decrementar el contador de comentarios de la noticia en 1 para reflejar solo comentarios activos. Sin embargo, en algunos diseños no se decrementa para mantener conteo histórico; dado que Menéame muestra el número de comentarios actuales, es razonable restarlo tras eliminación.
4. Tras estos cambios, el servidor confirma al Moderador la acción completada.
5. En la interfaz de todos los usuarios, ese comentario aparece removido (p. ej. una línea indicando "[comentario eliminado]").
6. Si el comentario eliminado era raíz de un hilo con muchas respuestas, normalmente las respuestas *no* se eliminan sino que quedan huérfanas con indicación del padre eliminado. Esto mantiene la jerarquía aunque una pieza esté ausente.
7. Menéame también podría registrar esta acción en la tabla de `logs` (p. ej. un log de moderación).

Moderación y Administración (descartar enlaces, banear usuarios)

Aquí se detallan dos acciones típicas de administración: descartar una noticia pendiente y banear (deshabilitar) a un usuario, incluyendo las interacciones con la base de datos.

a. Secuencia de *Descartar enlace:* Un administrador/moderador decide **descartar** (eliminar de la cola) una noticia pendiente por incumplir normas o spam, antes de que se publique.

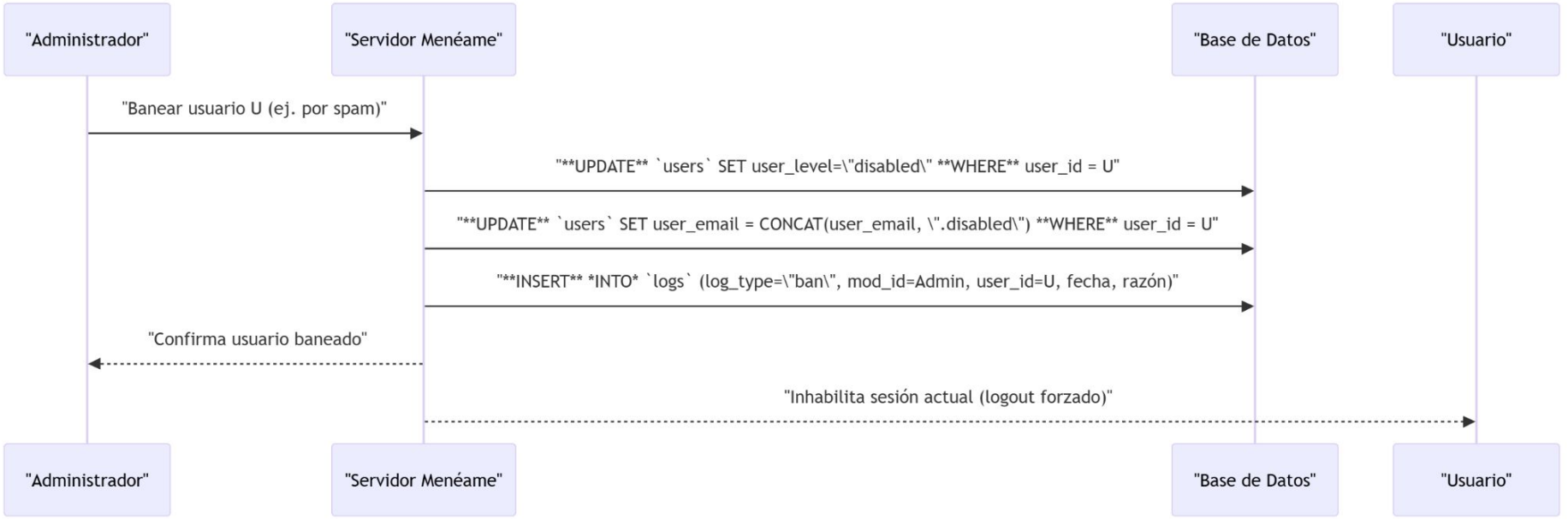
Moderación y Administración (descartar enlaces, banear usuarios)

Secuencia de *Descartar enlace*
En este flujo, un **Administrador** o moderador elige una noticia en pendientes (estado *queued*) y la descarta manualmente.

1. El **Servidor** al recibir esta orden actualiza el estado del enlace en la base de datos: `link_status` se establece a `"discard"`. Esto indica que la noticia ha sido rechazada manualmente. Si la noticia tenía un campo de fecha de publicación previsto, se pone a NULL (no se publicará).
2. Adicionalmente, se suele registrar el hecho en la tabla de logs para auditoría, incluyendo quién la descartó y posiblemente un motivo.
3. Desde ese momento, la noticia deja de aparecer en la lista de pendientes para los usuarios normales.
4. Menéame podría notificar al **autor** del enlace que su envío fue descartado (por ejemplo, mostrárselo en su feed personal de actividad o enviarle una notificación interna). Este paso es opcional pero útil; en caso de migración, se podría implementar vía un sistema de mensajes.
5. En la interfaz pública, la noticia descartada podría seguir visible en la sección de *descartadas* (Menéame muestra las descartadas durante un tiempo con un icono gris, indicando que no prosperaron), o simplemente desaparecer.
6. Cabe mencionar que Menéame distingue *discard* (descartada por moderación) de *autodiscard* (descartada automáticamente por falta de votos), pero ambos resultan en que la noticia no llega a portada.

Moderación y Administración (descartar enlaces, banear usuarios)

b. Secuencia de *Banear usuario:* Un administrador decide **deshabilitar** o banear a un usuario por mal comportamiento. Se cambia el estado del usuario en la base de datos para impedirle el acceso.



Moderación y Administración (descartar enlaces, banear usuarios)

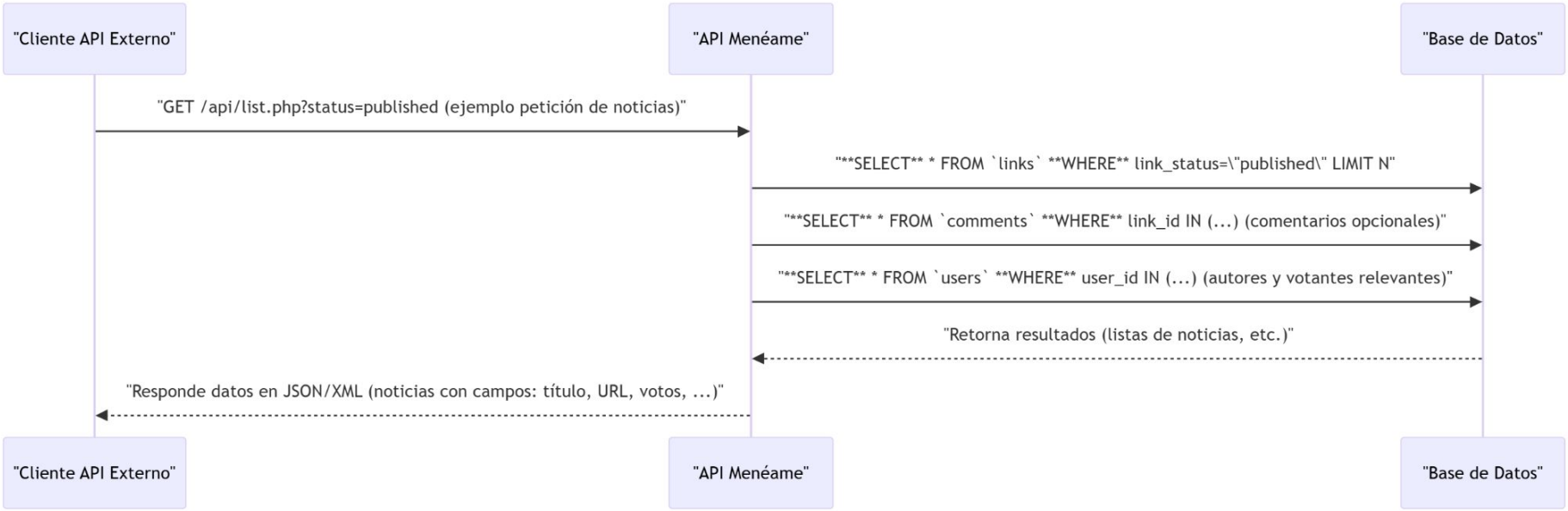
Secuencia de *Banear usuario*
Descripción: Para **banear a un usuario**, un **Administrador** ejecuta la acción de deshabilitar la cuenta.

1. El **Servidor** marca al usuario en la base de datos cambiando su nivel o estado. En Menéame, la columna `user_level` del usuario se establece a `"disabled"` (o `"autodisabled"` si fue una acción automática). Esto inmediatamente evita que esa cuenta pueda autenticarse: el código de autenticación comprueba si `user_level` es `'disabled'` y en tal caso fuerza logout
2. Adicionalmente, es práctica común *anonimizar* o modificar ciertos datos del usuario baneado, como su correo electrónico, para prevenir abuso o para marcarlo internamente. En la secuencia, se concatena ".disabled" al email (esto se deduce de la política de Menéame de añadir `@disabled` a emails de cuentas auto-deshabilitadas, práctica observada en el código de limpieza).
3. También podría bloquear su avatar u otros detalles.
4. Se inserta un registro en `logs` indicando que el usuario fue baneado, quién lo hizo y por qué, quedando constancia.
5. El sistema puede terminar cualquier sesión activa de ese usuario (por ejemplo, eliminando sus cookies de autenticación o invalidando sus tokens).
6. Desde la perspectiva del usuario baneado, cuando intente usar el sitio notará que su sesión ha caducado y no podrá volver a iniciar sesión (el sistema rechazará sus credenciales al encontrar el flag de disabled).
7. Cualquier contenido futuro de ese usuario (comentarios, envíos) es bloqueado. Sus contenidos pasados pueden permanecer pero asociados a un usuario deshabilitado.

Integración con API y Automatización

Menéame ofrece una API para acceder a datos públicamente y emplea scripts automáticos para tareas de mantenimiento (como el cálculo de karma periódico). Se presentan dos secuencias: una de consulta vía API externa, y otra de un proceso automático interno (recalculo de karma diario).

a. Secuencia de *Consulta API (externa):* Un sistema externo (ej. la app móvil de Menéame) solicita datos mediante la API pública. El servidor responde con los datos solicitados tras consultarlos en la base de datos.



Integración con API y Automatización

Secuencia de *Consulta API (externa)*

Descripción: Un **Cliente externo** (por ejemplo, la aplicación móvil oficial o un script de terceros) envía una petición HTTP a la **API de Menéame** solicitando cierta información. Menéame provee endpoints públicos (por ejemplo `/api/list.php`) para obtener las noticias en portada, pendientes, los comentarios de una noticia, etc.

En la secuencia de ejemplo, el cliente pide la lista de noticias publicadas.

1. El **Servidor** (API) procesa la solicitud consultando la **Base de Datos**: hace un **SELECT** de las noticias con `link_status="published"` (posiblemente con límites/paginación).
2. Puede también enriquecer la respuesta con datos relacionados: por ejemplo, incluir un resumen de comentarios, para lo cual podría consultar la tabla `comments` filtrando por esos links, y/o la tabla `users` para obtener los nombres de los autores de las noticias o comentarios.
3. Una vez obtenidos los datos necesarios, el servidor los formatea en la respuesta (generalmente JSON o XML). En nuestro caso, la API `/api/list.php` de Menéame devuelve un JSON con un array de noticias y sus campos principales (id, título, URL, número de votos, karma, autor, fecha, etc.).
4. El servidor envía esa respuesta al cliente externo, que la utiliza en su interfaz. Este proceso **no** involucra sesión de usuario (es lectura pública).
5. Si fuera una API con autenticación (por ejemplo, enviar un voto vía API), habría pasos de auth (pasar una clave API o token de usuario y validarlo).
6. El desempeño es importante: Menéame utiliza *SQL cache* y combinaciones de joins para optimizar estas consultas

Integración con API y Automatización

b. Secuencia de *Proceso automático (re-cálculo de karma diario):* Menéame ejecuta tareas programadas (cron) para mantenimiento. Un ejemplo crítico es el recalculo periódico del karma de usuarios en base a sus actividades (votos emitidos y recibidos, publicaciones, antigüedad, etc.), realizado por un script como `karma6.php`. A continuación se ilustra de forma simplificada:

Integración con API y Automatización

Secuencia de *Proceso automático (re-cálculo de karma diario)*

Descripción: Cada noche (o con la periodicidad configurada), un proceso **Cron** del servidor lanza el **script de karma** (un proceso PHP independiente, e.g. `karma6.php`). Este **Proceso automático** realiza varias consultas agregadas para recalcular las reputaciones.

1. Primero, calcula estadísticas globales que sirven en el algoritmo: por ejemplo el promedio de karma de enlaces publicados recientemente, para calibrar cuánto valor tiene publicar una noticia en ese periodo; también puede contar cuántas noticias fueron descartadas, etc.
2. Luego, para cada usuario, se evalúa su actividad: cuántos votos positivos/negativos ha dado a noticias que terminaron publicadas o descartadas, cuántas noticias ha enviado y de esas cuántas se publicaron, cuántos comentarios, etc. Esto implica consultas que unen la tabla de `votes` con `links` y filtran por distintos criterios (por ejemplo, en el código se ven consultas que cuentan votos de un usuario en enlaces publicados vs. descartados, posiblemente para ajustar su karma).
3. Con esos datos, el script aplica una fórmula para cada usuario. Simplificando, el karma de usuario sube con noticias publicadas, votos positivos dados a contenidos que luego son bien valorados, antigüedad, etc., y baja con comportamientos como muchos envíos que acaban descartados, exceso de votos negativos injustificados, o inactividad prolongada.
4. El resultado final es un nuevo valor de `user_karma` para cada usuario.
5. El script entonces realiza **UPDATEs** en la tabla `users` asignando el nuevo karma a cada usuario. También aplica ciertas reglas directas: por ejemplo, fija el karma mínimo para cuentas *baneadas* – el código establece karma = 6 para usuarios con `user_level` "disabled" si tenían más alto, limitando la influencia de cuentas bloqueadas.
6. Tras recalcular karmas, el script suele realizar tareas de **limpieza**: por ejemplo, borrar logs antiguos (entradas de la tabla `logs` de hace más de X días), eliminar usuarios no validados tras cierto tiempo, o purgar enlaces descartados antiguos para liberar espacio(excepto los de tipo "article" que se conservan más tiempo).
7. Una vez completadas las operaciones, el proceso termina. Menéame puede enviar un breve resumen a los administradores o simplemente guardar en un log de sistema.
8. Este **Diagrama de Flujo de Datos** del proceso de karma muestra cómo el script actúa como un usuario interno con altos privilegios: realiza lecturas masivas de varias tablas y luego escribe en la tabla de usuarios y otras para mantener la base de datos sana y calculada. Este flujo de automatización asegura que la plataforma se mantenga estable (no sobrecargada de datos viejos) y que el **karma** refleje continuamente las aportaciones de los usuarios de forma justa.

Diagrama de Flujo de Datos (DFD)

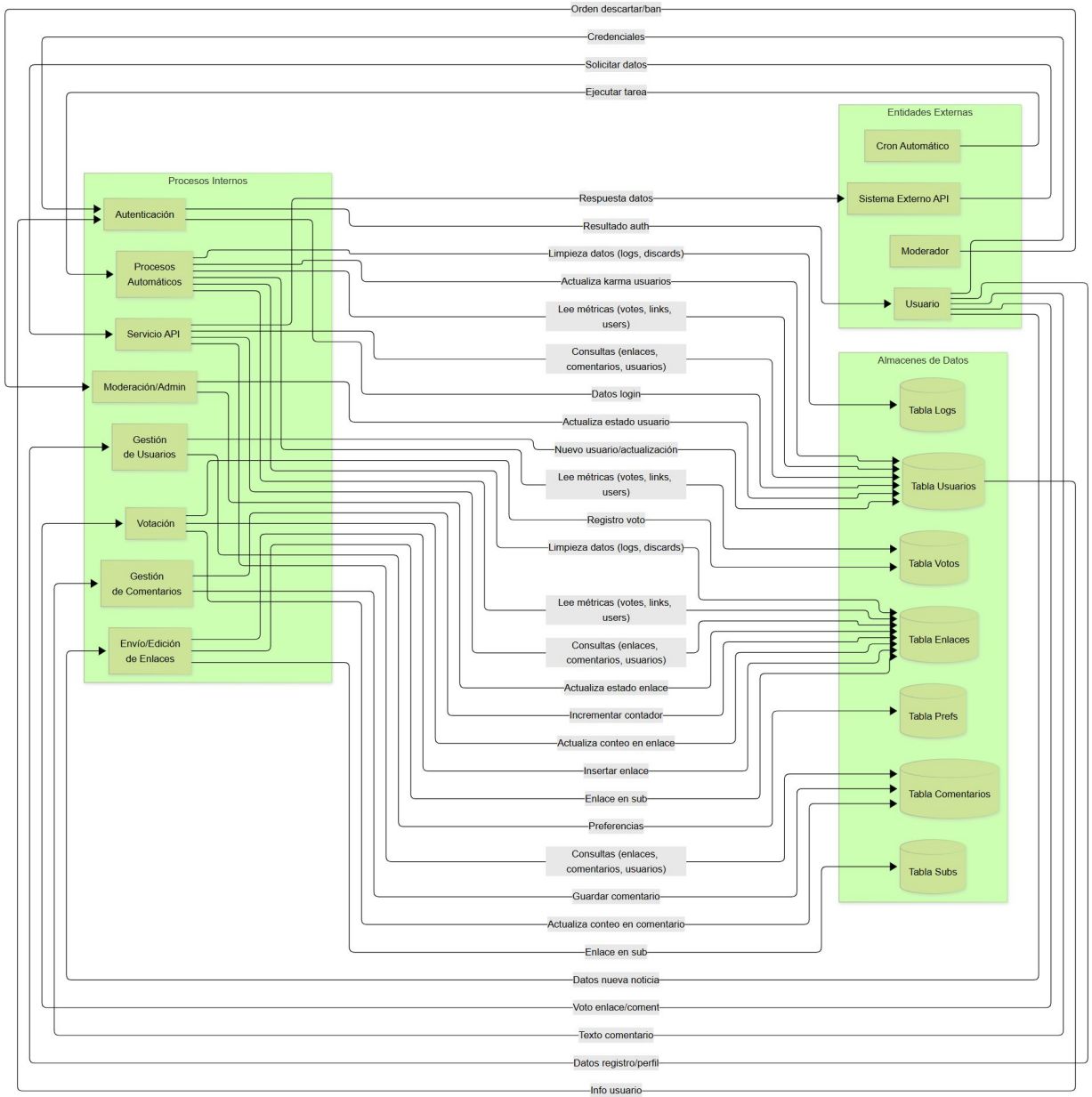


Diagrama de Flujo de Datos (DFD)

Descripción del DFD: Este **Diagrama de Flujo de Datos** resume las interacciones entre los actores, procesos y datos del sistema:

- El **Usuario** (externo) interactúa con procesos como *Autenticación* (enviando credenciales para login) y *Gestión de Usuarios* (al registrarse o actualizar su perfil), *Envío de Enlaces* (cuando envía una noticia), *Votación* (al votar contenidos) y *Gestión de Comentarios* (al publicar un comentario). Cada uno de estos procesos toma los datos del usuario y realiza operaciones en los **almacenes de datos** correspondientes: por ejemplo, Autenticación verifica los datos en la tabla **Usuarios**, Votación inserta un registro en **Votos** y actualiza los contadores en **Enlaces** o **Comentarios**, etc.
- El **Moderador/Admin** (otro actor externo) interacciona principalmente con el proceso de *Moderación/Admin*, ordenando acciones como descartar un enlace o banear un usuario. Este proceso a su vez modifica los almacenes de **Enlaces** (cambiando el estado de una noticia) o **Usuarios** (cambiando el nivel o estado de un usuario), y puede añadir registros a **Logs** (aunque por simplicidad no se dibujó cada log, en la implementación sí ocurre registro de estos eventos en DLogs).
- Un **Sistema Externo** (cliente de API) envía solicitudes al proceso *Servicio API*. Este proceso realiza lecturas en los almacenes de datos relevantes (principalmente **Enlaces**, **Comentarios**, **Usuarios** – y potencialmente también **Votos** si se incluyen conteos) y devuelve la información solicitada al sistema externo en formato consumible (JSON/XML). No modifica datos, solo lectura.
- El **Cron Automático** activa periódicamente procesos automáticos (aquí representado por *Procesos Automáticos*, e.g. el script de karma). Este proceso interno lee grandes volúmenes de datos de **Votos**, **Enlaces**, **Usuarios**, etc., los procesa, y luego escribe resultados a la base (actualiza karmas en **Usuarios**, limpia registros en **Logs**, elimina enlaces descartados de **Enlaces** según reglas, etc.). En el diagrama, por claridad, se muestra lectura de Votos/Enlaces/Usuarios y actualización de Usuarios/Logs/Enlaces como flujos representativos.
- Los **Almacenes de Datos** (tablas) están todos dentro del mismo sistema de base de datos MySQL. Aunque en DFD se dibujan separados por concepto, en la realidad forman parte de la misma base *meneame*. Las flechas hacia o desde las tablas implican consultas SQL (SELECT) o modificaciones (INSERT/UPDATE/DELETE).

Diagrama de Clases

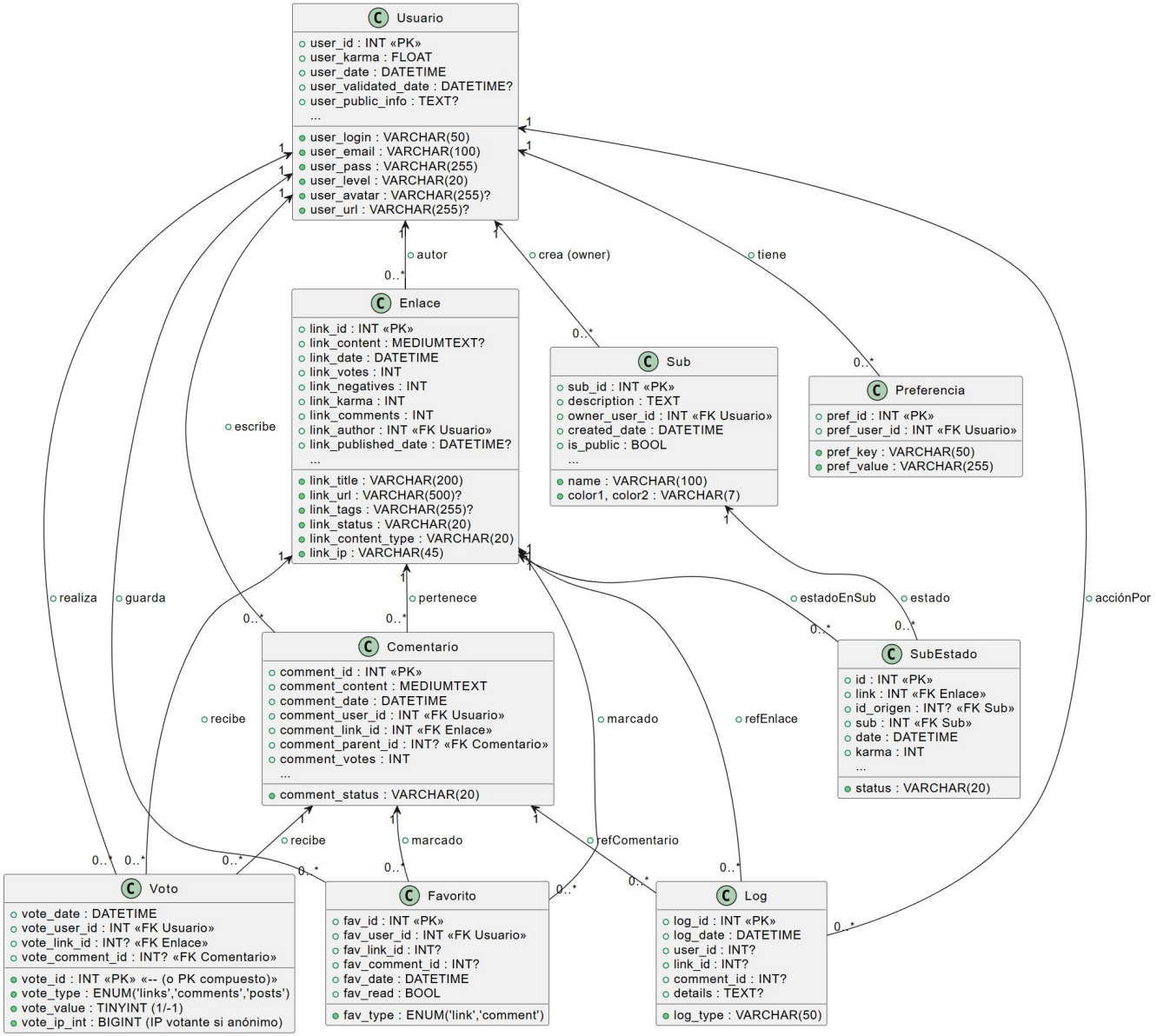


Diagrama de Clases

Usuario
• *Identificación:* PK user_id
• *Atributos clave:* user_login, user_email, user_pass (hash), user_level, user_karma, fechas de registro y validación, datos de perfil (avatar, URL, bio, nombre)
• *Relaciones:*
  – Envía muchos Enlaces
  – Es autor de numerosos Comentarios
  – Realiza múltiples Votos, guarda varios Favoritos y tiene diversas Preferencias
  – Puede crear/participar en Subs y aparece en Logs (ej. baneos)

Enlace
• *Identificación:* PK link_id
• *Atributos clave:* Título, URL (o contenido interno), etiquetas, fecha, estado (queued, published, etc.), tipo (link, article)
• *Contadores:* votos, negativos, karma, comentarios, clics
• *Relaciones:*
  – Pertenece a un Usuario (autor)
  – Acumula Comentarios, Votos, Favoritos y registros en Logs
  – Se asocia a Subs mediante SubEstado

Comentario
• *Identificación:* PK comment_id
• *Atributos clave:* Contenido, fecha, posible puntuación y estado (activo, borrado)
• *Relaciones:*
  – Relacionado con un Usuario y un Enlace
  – Puede responder a otros Comentarios (jerarquía)
  – Recibe Votos y puede ser marcado como Favorito; se registra en Logs

Voto
• *Identificación:* PK vote_id (o compuesto)
• *Atributos clave:* Tipo (links, comments, posts), valor (1 o -1), fecha y, para anónimos, IP
• *Relaciones:*
  – Vinculado a Usuario, Enlace o Comentario (según tipo)

Sub (Comunidades)
• *Identificación:* PK sub_id
• *Atributos clave:* Nombre, descripción, propietario, fecha de creación y configuración (colores, privacidad)
• *Relaciones:*
  – Cada Sub tiene varios estados de enlace a través de SubEstado

SubEstado
• *Función:* Tabla intermedia que vincula Enlaces y Subs
• *Atributos clave:* Estado del enlace en la sub (queued, published), fecha, karma específico y posible referencia de origen

Favorito
• *Identificación:* PK favorite_id
• *Atributos clave:* Tipo (link o comment), referencia al contenido (favorite_link_id o favorite_comment_id), timestamp y estado de lectura
• *Relaciones:*
  – Pertenece a un Usuario y se asocia a un Enlace o Comentario

Preferencia
• *Identificación:* PK pref_id
• *Atributos clave:* Clave (ej. "subs_default", "comment_order") y valor
• *Uso:* Configuración personalizada del usuario (adaptable en Django)

Log
• *Identificación:* PK log_id
• *Atributos clave:* Tipo de evento (ej. "comment_new", "login_failed", "ban", "link_discard"), fecha y detalles adicionales
• *Relaciones:*
  – Registra acciones que involucran Usuarios, Enlaces y Comentarios para monitoreo y estadísticas

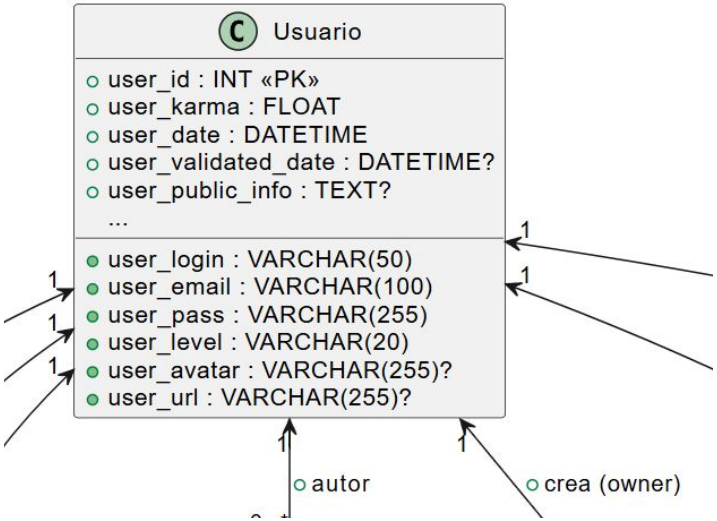
Resumen de Relaciones Generales:
– Un Usuario puede tener múltiples Enlaces, Comentarios, Votos, Favoritos y Preferencias.
– Cada Enlace, publicado por un Usuario, puede recibir muchos Comentarios y Votos, y ser marcado como Favorito.
– Los Comentarios pueden formar hilos (padre-hijo) y acumular Votos y Favoritos.
– La asociación de Enlaces con Subs se gestiona mediante SubEstado, permitiendo estados independientes en cada comunidad.
– Los Logs documentan eventos relevantes de todas las entidades para seguimiento y análisis.

Diagrama de Clases

Usuario (`users`): Representa a los usuarios registrados. Tiene como clave primaria `user_id`. Entre sus atributos principales están:

- `user_login` (nombre de usuario),
- `user_email`, `user_pass` (contraseña almacenada con hash),
- `user_level` (nivel/rol, e.g. *normal*, *admin*, *god*, *disabled*, *autodisabled*),
- `user_karma` (karma actual del usuario, numérico), `user_date` (fecha de registro),
- `user_validated_date` (fecha de validación de email, puede ser nula hasta que confirma su correo).

Otros campos almacenan información de perfil: `user_avatar` (ruta o ID del avatar), `user_url` (web personal), `user_public_info` (bio o información pública del usuario), `user_names` (nombre real), etc.



Las relaciones:

- **Usuario – Enlace** es de uno a muchos: un usuario puede haber enviado *muchos* enlaces/noticias, cada enlace tiene un único autor (fk `link_author` a `user_id`).
- **Usuario – Comentario** es 1:N (un usuario escribe muchos comentarios).
- **Usuario – Voto**: un usuario puede realizar muchos votos (relación 1:N desde Usuario a Voto).
- También un usuario puede tener muchos **Favoritos** guardados (relación 1:N Usuario–Favorito),
- y varias **Preferencias** (relación 1:N Usuario–Preferencia).
- Si hay subs (comunidades), un usuario podría poseer (ser owner de) una o varias comunidades (**Sub**), o participar en ellas (esa participación no se modela directamente aquí porque es abierta según nivel de karma en Menéame).
- Además, un usuario puede aparecer referenciado en **Log** para indicar acciones que realizó (por ejemplo, un log de tipo "ban" tendrá el admin en user_id y el baneado posiblemente en otro campo).

Diagrama de Clases

Enlace (`links`): Representa una noticia/enlace enviado. `link_id` es la PK. Atributos principales incluyen:

- `link_title` (título de la noticia),
- `link_url` (URL del contenido si es externo; si es un *post* interno, este campo puede ser nulo y el contenido estará en `link_content`),
- `link_content` (texto del cuerpo o resumen, usado en artículos o descripciones),
- `link_tags` (etiquetas de la noticia separadas por comas),
- `link_date` (fecha/hora de envío),
- `link_status` (estado del enlace: típicamente "queued" para pendientes, "published" para publicadas, "discard" para descartadas manualmente, "autodiscard" para descartadas por el sistema, "abuse" para marcadas por spam),
- `link_content_type` (tipo de contenido: "link" o "article", incluso podría haber "quiz" u otros si existieran tipos especiales; en Menéame se usa para distinguir si es un enlace externo o un *text post*).

También tiene múltiples campos contadores calculados: `link_votes` (número de votos positivos recibidos), `link_negatives` (votos negativos recibidos), `link_karma` (puntuación calculada de la noticia, combinando votos +/-, posiblemente afinado por algoritmo), `link_comments` (cantidad de comentarios recibidos), `link_clicks` (si se habilita contador de clics externos). El campo `link_author` es una FK al Usuario que envió la noticia. `link_published_date` almacena cuándo pasó a portada (o NULL si no publicada). También hay campos como `link_ip` (IP del remitente) y `link_thumb_status` (estado de miniatura).

- **Enlace – Comentario** es 1:N: una noticia tiene muchos comentarios asociados (comentarios apuntan a link_id).
- **Enlace – Voto** también es 1:N: una noticia puede tener muchos votos (vote_link_id apunta a link_id).
- **Enlace – SubEstado**: esta relación modela la pertenencia de un enlace a una comunidad (*sub*). Menéame permite enviar a la cola general o a comunidades específicas, incluso la misma noticia puede estar en la general y ligada a una sub comunidad (con origen en subs). La tabla `sub_statuses` parece hacer un seguimiento de la relación enlace-subs: en el diagrama, *SubEstado* tiene FK a Enlace y a Sub, más un campo `status` (estado del enlace dentro de esa sub, p. ej. queued o published, independiente del estado global), y posiblemente un campo `id_origen` para saber si la noticia vino de otra sub (campo origen). Para la migración, esto sugiere un modelo intermedio (ManyToMany) entre *Link* y *Sub* con campos extra (estado, karma en la sub, fecha).
- **Enlace – Favorito**: un enlace puede ser marcado favorito por muchos usuarios, de ahí 1:N (cada favorito de tipo link tiene link_id apuntando al enlace).
- **Enlace – Log**: acciones sobre un enlace (descartes, publicaciones auto) pueden registrarse en logs.

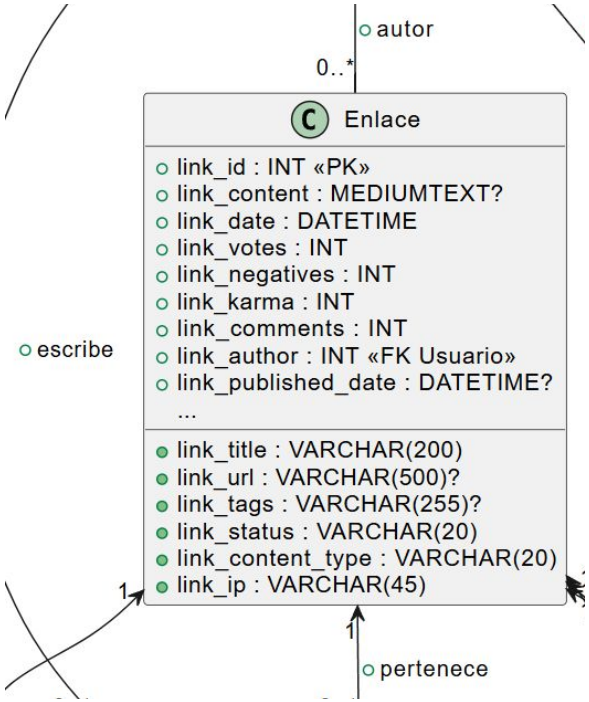


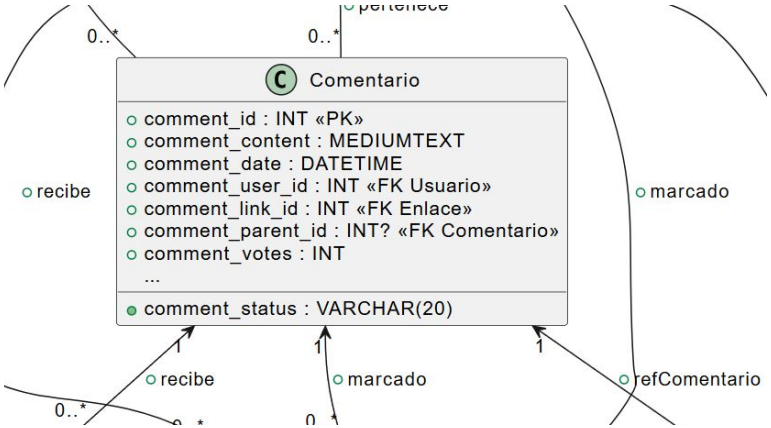
Diagrama de Clases

Comentario (`comments`): Representa comentarios de los usuarios en noticias. `comment_id` PK. Tiene

- `comment_content` (texto del comentario),
- `comment_date` (fecha/hora),
- y FK `comment_user_id` (autor, ref a Usuario),
- `comment_link_id` (la noticia en la que se comenta, ref a Enlace).

Para soportar la jerarquía, `comment_parent_id` referencia opcionalmente a otro comentario (self-FK) indicando a quién responde; si es NULL, es un comentario raíz del hilo de la noticia. Puede haber campos de conteo o score: en la base actual no se observó explícito, pero hemos incluido `comment_votes` o un `comment_score` para reflejar la puntuación neta (podría calcularse sobre la marcha sumando votos de la tabla `votes` donde comment_id=Y y value positivo/negativo).

También un campo `comment_status` para marcar si fue borrado u oculto.

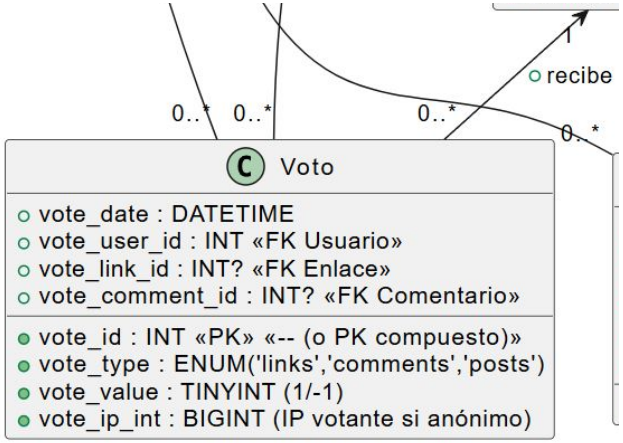


- **Comentario – Usuario** (N:1, cada comentario tiene un autor usuario),
- **Comentario – Enlace** (N:1, cada comentario pertenece a una noticia).
- **Comentario – Comentario** (self-relación 0..* a 1, cada comentario puede tener respuestas hijas).
- **Comentario – Voto**: similar a enlaces, un comentario puede recibir muchos votos (vote_comment_id apunta a comment_id).
- **Comentario – Favorito**: los usuarios también pueden guardar comentarios como favoritos, relación 1:N (un comentario puede estar en varios favoritos de distintos usuarios).
- **Comentario – Log**: si hay registros de moderación de comentarios eliminados, podrían aparecer en logs con comment_id referenciado.

Diagrama de Clases

Voto (`votes`): Registra cada voto emitido por usuarios. Dependiendo de implementación, la PK podría ser un id artificial `vote_id` o la combinación (usuario, tipo, id_objeto) para evitar duplicados. Aquí lo modelamos con `vote_id` PK para claridad. Campos:

- `vote_type` indica sobre qué objeto es el voto (posibles valores: "links", "comments", "posts"),
- `vote_value` es normalmente 1 o -1 (Menéame los denomina *menéos* y *votos anónimos negativos* para noticias, y probablemente similar en comentarios),
- `vote_date` fecha del voto. Luego,
- `vote_user_id` FK al Usuario que vota (o 0/NULL si es un voto anónimo, aunque en código se ve manejo de IPs para anónimos),
- `vote_link_id` FK al Enlace si aplica,
- `vote_comment_id` FK al Comentario si aplica. (En vez de dos columnas separadas, Menéame podría tener una sola columna `vote_link_id` y reutilizarla para comentarios con IDs especiales, pero por las consultas parece que usan columnas separadas: en la JOIN de votos para links se filtra `vote_type="links"`, lo que sugiere que vote_link_id se usa solo cuando type=links. Para seguridad en el modelo, lo hemos separado.)
- Adicionalmente, Menéame guarda la IP numérica (`vote_ip_int`) del votante si es no autenticado (así evita múltiples votos de un mismo IP no logueado).



Relaciones:

- **Voto – Usuario** (muchos votos pertenecen a un usuario, o a "anónimo"),
- **Voto – Enlace** (muchos votos asociados a una noticia, filtrados por type="links"),
- **Voto – Comentario** (muchos votos asociados a un comentario, con type="comments").

Diagrama de Clases

Sub (`subs`): Representa una comunidad o *sub* dentro de Menéame (una sección temática creada por usuarios). Campos principales:

- `id` (PK, aquí llamado sub_id),
- `name` (nombre de la comunidad),
- `description`,
- `owner_user_id` (FK al usuario creador/administrador de la sub),
- fechas de creación, posiblemente flags como si es pública o privada, colores de tema (`color1`, `color2`) para personalización de la página, etc.

Relaciones:

- **Sub – SubEstado**: una sub tiene muchos estados de enlaces (entradas en la cola de esa sub).
- **Sub – Usuario**: un usuario puede crear una sub (owner), y posiblemente haya relación de moderadores, pero en Menéame el creador es el admin principal de la sub.

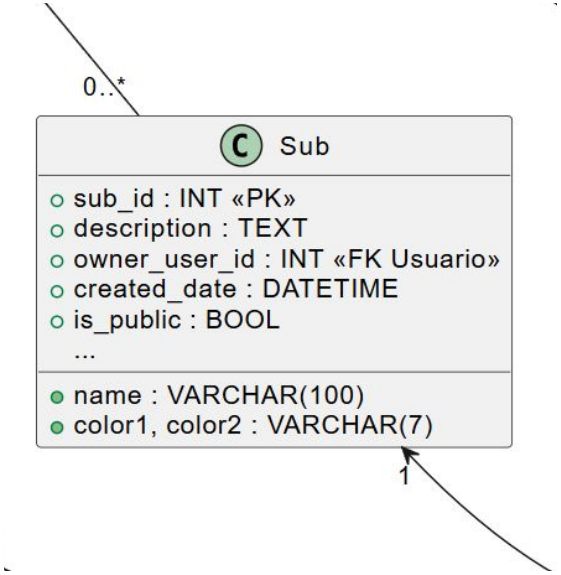


Diagrama de Clases

SubEstado (`sub_statuses`): Esta tabla vincula *enlaces* con *subs*, almacenando el estado de un enlace en una sub. Tiene un FK `link` (al enlace), FK `sub` (a la sub), y un campo `status` (p.ej. "published" o "queued" dentro de esa sub).

Aparece también un campo `id` (PK) y posiblemente `id_origen` o `origen` para señalar la sub de origen si el enlace fue movido desde otra sub o promovido al global (en el código se ve `creation.id = creation.origen` y `creation.id = subs.id`, lo que sugiere que *origen* puede ser una referencia a la sub original de la cual se envió el enlace).

También tienen `date` y quizá `karma` específico dentro de la sub.

Esta estructura es un poco compleja; esencialmente modela una relación *many-to-many* entre Enlace y Sub con metadata. En el diagrama, la clase SubEstado sirve de tabla intermedia: un Enlace puede tener múltiples SubEstado (ej. si está en el global y en una sub, o en múltiples subs aunque creo que un enlace solo puede estar en una sub o global), y cada Sub puede tener múltiples enlaces asociados.

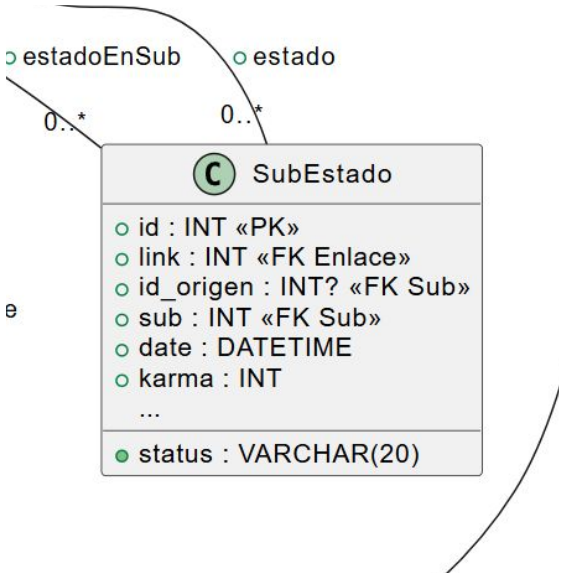
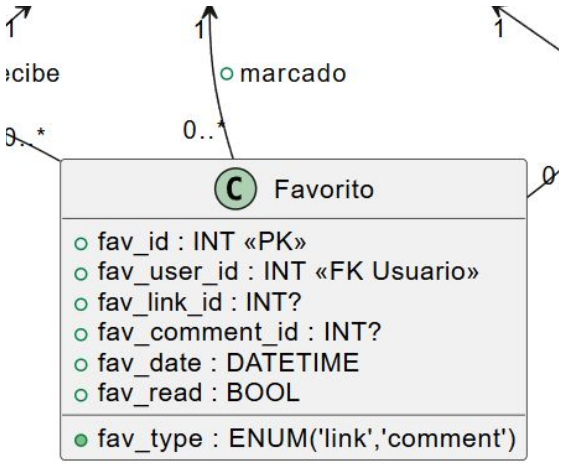


Diagrama de Clases

Favorito (`favorites`): Tabla que guarda contenido favorito guardado por usuarios. Tiene:

- `favorite_id` (PK),
- `favorite_user_id` (FK Usuario),
- `favorite_type` ("link" o "comment"), y según el tipo,
- `favorite_link_id` o `favorite_comment_id`.
- También un timestamp y posiblemente un campo `favorite_read` (Menéame marca si el usuario ya leyó la conversación desde su última visita). Esta tabla permite a un usuario marcar noticias o comentarios para leer después o seguir. En la clase Favorito del diagrama, se maneja similar a Voto con campos para link/comment.



Relaciones:

- **Favorito – Usuario** (N:1, cada favorito pertenece a un usuario),
- **Favorito – Enlace/Comentario** (N:1 según el tipo, apunta al contenido marcado).

Diagrama de Clases

Preferencia (`prefs`): Tabla de preferencias de usuario (clave-valor).
Campos:

- `pref_id` (PK),
- `pref_user_id` (FK Usuario),
- `pref_key` (ej. "subs_default", "comment_order"),
- `pref_value`. En el código, por ejemplo, vimos su uso para almacenar la subscripción por defecto del usuario

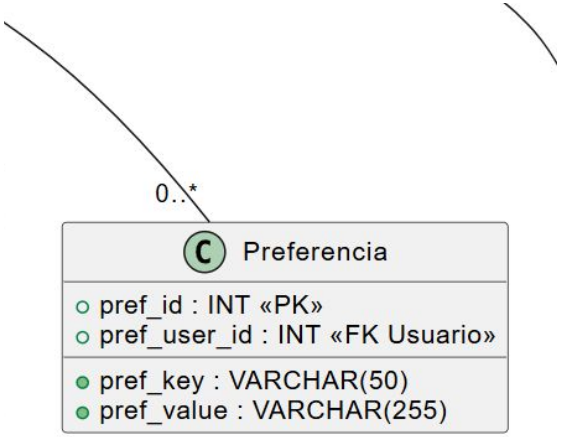


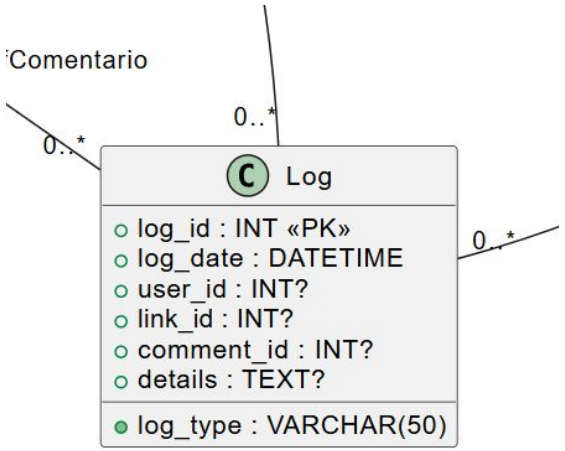
Diagrama de Clases

Log (`logs`): Registra eventos importantes para monitoreo y estadísticas: ej. logins fallidos, comentarios nuevos (para el *fisgón* en tiempo real), baneos, acciones admin, etc.

Campos:

- `log_id` PK,
- `log_type` (tipo de evento, e.g. "comment_new", "login_failed", "ban", "link_discard"...),
- `log_date`, y campos opcionales referenciando entidades involucradas:
- `user_id`,
- `link_id`,
- `comment_id`,
- además de quizá un campo `details` o `data` con información extra (IP, motivo, etc.). Vimos en el script que se limpian logs antiguos de tipos específicos.

Esta clase se relaciona con casi todas: puede tener una referencia a Usuario (usuario que hizo algo o el afectado), a Enlace, a Comentario según el evento.



Conclusiones

El análisis exhaustivo de la arquitectura de Menéame y sus flujos (autenticación, publicación de enlaces, comentarios, votaciones, moderación, etc.) pone de manifiesto diversos desafíos:

Acoplamiento elevado entre la lógica de negocio y la integración de datos

Actualmente, Menéame combina la lógica de negocio (PHP legacy) con procesos de integración y manipulación de datos (scripts, consultas directas a la base de datos, etc.), lo que dificulta la mantenibilidad y la incorporación de mejoras. El monolito PHP dificulta la división de la aplicación y la escalabilidad segmentada.

Necesidad de escalabilidad y flexibilidad

Menéame debe procesar grandes volúmenes de datos y adaptarse a cambios de forma ágil, en especial al integrar múltiples orígenes de información (servicios externos, migraciones históricas, lotes de datos, nuevos servicios, campañas, etc.).

Control de flujos y orquestación ante nuevas necesidades

En la situación actual, se depende de scripts internos y tareas programadas (cron jobs), lo cual dificulta la supervisión y la incorporación de funcionalidades (algoritmos de karma, automatizaciones de moderación, etc.).

Procesamiento asíncrono y tolerancia a cargas pico

Las tareas intensivas (horas punta de votos, envíos masivos de enlaces, etc.) siguen basándose en consultas síncronas y cron jobs, lo que puede derivar en bloqueos o retrasos críticos.

Evolución de una Aplicación Monolítica a una Arquitectura Basada en Mensajería

- **Exponer Funcionalidad vía APIs:**
  Convertir los procesos principales (registro de usuarios, envío de enlaces, gestión de comentarios, votos, etc.) en **endpoints REST**, de modo que el backend se limite a las reglas de negocio y la validación de las peticiones. Con ello, se logra un núcleo más sencillo de mantener y evolucionar.
- **Publicar Eventos en el Broker:**
  Tras cada operación relevante (nuevo enlace, voto registrado, comentario), el backend envía un **mensaje** al broker, describiendo el suceso. Gracias a este modelo, se fomenta la comunicación desacoplada y la posibilidad de añadir nuevos consumidores sin modificar la lógica base.
- **Consumir Mensajes para Orquestar Procesos:**
  Módulos o microservicios específicos (servicio de karma, moderación, notificaciones, etc.) se suscriben a los mensajes del broker para ejecutar acciones. Por ejemplo, una votación dispara la actualización de karma, o un comentario nuevo desencadena notificaciones a usuarios suscritos.
- **Escalado y Observabilidad:**
  Mediante la configuración y monitorización de colas y tópicos, es posible ajustar la **capacidad de procesamiento** añadiendo o quitando consumidores según la demanda. Además, se dispone de métricas de tráfico, alertas en tiempo real y reintentos automáticos si ocurre algún error.

Conclusión

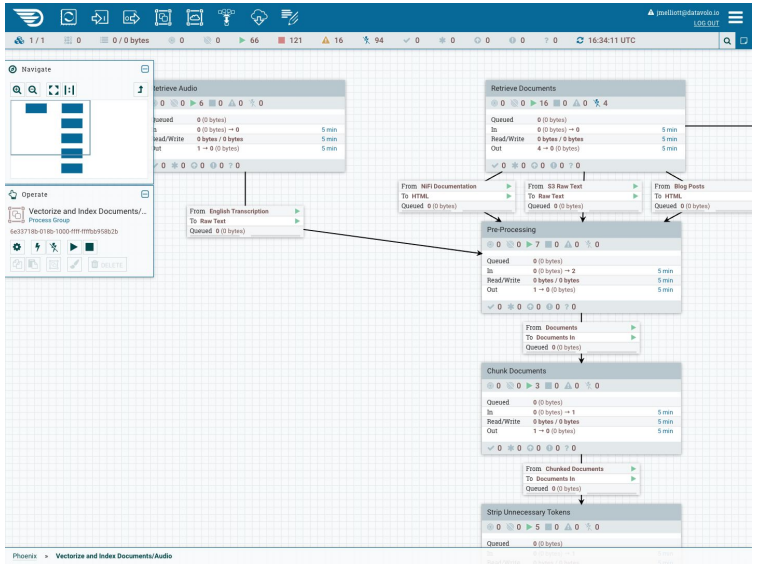
Adoptar un **paradigma de mensajería** con un **broker abierto** ofrece a Menéame la posibilidad de **desacoplar la lógica de negocio**, **ampliar la escalabilidad** y habilitar el **procesamiento asíncrono** de flujos. De este modo, el sistema evoluciona hacia una **arquitectura modular y flexible**, en la que cada servicio se centra en su propia funcionalidad y se comunica mediante eventos, logrando mayor **control** sobre las cargas de trabajo y las integraciones de datos. Sumado a ello, la incorporación de **Apache NiFi** como capa de orquestación proporciona **fluidez** en la gestión de flujos, escalabilidad en la ejecución de procesos y un seguimiento detallado que facilita la resolución de incidencias y la rápida adopción de nuevas funcionalidades.

Propuesta new core

Por qué NiFi Puede Ser la Solución
Para orquestar estos flujos de datos, **Apache NiFi** se presenta como una alternativa muy potente. NiFi ofrece un **entorno gráfico** de diseño de flujos, gestión de conexiones y transformaciones, permitiendo:

- **Desacoplar** los orígenes y destinos de datos (el broker, la base de datos, servicios externos) sin sobrecargar el backend.
- **Automatizar** la ingesta y el procesamiento de eventos, delegando la lógica de integración a NiFi y liberando a Menéame para centrarse en la funcionalidad core.
- **Escalar** de forma flexible, al poder ejecutar NiFi en clúster y manejar grandes volúmenes de datos o procesos simultáneos.
- **Monitorizar y Reintentar** los flujos con facilidad, gracias a sus herramientas de seguimiento en tiempo real, lo que facilita la localización de cuellos de botella y la corrección rápida de errores.

En resumen, NiFi no solo simplifica la orquestación y el enrutamiento de eventos en una arquitectura de mensajería, sino que también aporta **visibilidad**, **modularidad** y **resiliencia** a la solución final.



Apache NiFi

Framework con Apache NiFi

1. **Orquestación Visual de Flujos**
   NiFi proporciona una interfaz gráfica que facilita la configuración y gestión de flujos de datos. Esto permite a los equipos técnicos y de negocio **visualizar** y **modificar** los procesos sin tener que bucear en el código de la aplicación, acelerando la implementación de nuevas integraciones o cambios en los procesos.
2. **Facilidad para Integraciones Externas**
   Menéame (o cualquier otra aplicación web) puede exponer su lógica mediante APIs. NiFi se integra con esos endpoints, pero también con **toda clase de sistemas externos** (bases de datos, colas de mensajería, servicios en la nube, etc.). Esto hace que la orquestación de datos sea más ágil, evitando el desarrollo de integraciones personalizadas en el propio framework.
3. **Procesamiento Asíncrono y Escalable**
   Muchos pasos (cálculos de karma, limpieza de registros, scrappeo de enlaces, etc.) pueden ejecutarse de forma asíncrona. NiFi gestiona colas internas y puede escalar horizontalmente (en un clúster), **evitando** sobrecargar la aplicación web. Así, se reparte la carga entre el backend (dedicado al servicio web y lógica central) y NiFi (dedicado al procesamiento intensivo de datos).
4. **Monitorización y Control de Errores en Tiempo Real**
   NiFi incluye **herramientas de seguimiento** de cada paso del flujo (historial detallado, alertas automáticas, reintentos, etc.). Esto mejora la capacidad de identificar cuellos de botella y puntos de fallo, así como **reprocesar** datos en caso de errores (sin necesidad de modificar el código de la aplicación).
5. **Menor Acoplamiento de la Lógica de Negocio**
   Al externalizar la orquestación y ciertos flujos de datos en NiFi, el **código de la aplicación** (PHP, Django u otro lenguaje) queda más **limpio** y enfocado en las reglas de negocio básicas. Cambios en los flujos de integración, transformaciones de datos o notificaciones se pueden gestionar en NiFi sin tocar el core de la aplicación.
6. **Rapidez en la Evolución y Mantenibilidad**
   Con NiFi, agregar nuevas entradas y salidas de datos (por ejemplo, conectar con otro servicio externo para procesamiento de texto o para notificaciones push) es mucho más rápido que desarrollar los mismos flujos en el framework. Esto aumenta la **flexibilidad** y la **velocidad** de respuesta a requerimientos futuros.
7. **Seguridad y Gobernanza**
   NiFi ofrece **versionado** de flujos, control de acceso a nivel de usuarios y grupos, y cifrado de datos en tránsito. Esto facilita la implantación de **buenas prácticas de seguridad** y auditoría del movimiento de datos. También separa lógicamente la gestión de credenciales y conexiones externas, favoreciendo la gobernanza de datos (quién accede a qué, qué transformaciones se aplican).

Énfasis en Escalabilidad, Distribución y Nuevos Flujos Basados en IA

En el enfoque recomendado para transformar Menéame, conviene **expandir** las consideraciones de escalabilidad y distribución de procesos, especialmente al incorporar **nuevos flujos de datos** que involucren tratamientos avanzados, como la detección automatizada de violaciones de términos de uso mediante **agentes de Inteligencia Artificial (IA)**.

1. **Arquitectura Basada en APIs y NiFi Distribuido**
   - Mantener la aplicación de Menéame (ya sea en PHP, Django u otro framework) como **proveedor de APIs** y lógica básica de negocio.
   - **NiFi** orquestaría la ingestión y procesamiento de datos con un **diseño distribuido** (entorno cluster) para enfrentar picos de tráfico, grandes volúmenes de contenido y análisis en tiempo real.
   - Al permitir que NiFi ejecute múltiples instancias (nodos) en clúster, se reparte la carga de trabajo y se incrementa la **tolerancia a fallos**.
2. **Incorporación de Agentes de IA para la Moderación y Clasificación**
   - La ventaja de NiFi es que puede **invocar** servicios externos (microservicios o contenedores Docker con IA, librerías locales de análisis, APIs de terceros, etc.) sin modificar el core de Menéame.
   - Para la **detección temprana** de violaciones de términos, NiFi canalizaría el contenido (por ejemplo, texto de comentarios, noticias, usuarios recién registrados) hacia un **agente de IA** que analice patrones de spam, discurso de odio, contenido inapropiado, etc.
   - Tras un primer veredicto (e.g. puntuación de riesgo), NiFi devuelvería el resultado a la aplicación o a un **módulo de moderación** que aplique las políticas definidas (marcar contenido para revisión humana, eliminar automáticamente, etc.).
   - Esta integración se realiza de forma **asíncrona** y escalable: es fácil añadir nuevos agentes IA para distintos idiomas, temáticas o tipos de análisis, y NiFi decide enrutarlos según la categoría de contenido.
3. **Escalabilidad Vertical y Horizontal de Flujos IA**
   - NiFi facilita el **rápido ajuste** de su caudal de datos: si el tráfico y la cantidad de contenido a analizar suben, se agregan más nodos NiFi para mantener un rendimiento óptimo.
   - Por su parte, los agentes de IA pueden estar **dockerizados** y replicados en un orquestador (Kubernetes, Docker Swarm, etc.), recibiendo peticiones desde NiFi según capacidad.
   - Esto evita sobrecargar el backend de Menéame, que así se concentra en su propia lógica de negocio (registro de usuarios, publicación de enlaces, etc.).
4. **Centralización de la Orquestación y Nuevos Flujos**
   - Además de la moderación asistida por IA, NiFi puede encargarse de **otras tareas** crecientes:
     - **Integración de datos** (por ejemplo, recopilar noticias externas o feeds que se quieran procesar antes de publicarlos).
     - **Análisis en tiempo real** de tráfico, votaciones, tendencias (alimentando un modelo de IA que detecte patrones anómalos).
     - **Notificaciones** (a Slack, Discord, correo, etc.) cuando un agente IA detecta un contenido potencialmente peligroso.
   - Cada flujo nuevo se configura gráficamente en NiFi, minimizando el impacto en el resto del sistema y permitiendo a equipos multidisciplinares (desarrolladores, data scientists, analistas) colaborar sin invadir el código principal de Menéame.
5. **Governanza, Monitorización y Evolución**
   - La **monitorización detallada** de NiFi (visualización de cada paso y su estado) otorga un alto grado de control sobre la orquestación. Si un agente IA falla o requiere actualización de modelo, es sencillo desacoplarlo y redirigir los flujos a otro servicio temporal.
   - La **seguridad** se refuerza centralizando las conexiones y credenciales en NiFi, junto con un versionado de cada cambio en los flujos. Esto sirve de auditoría y evita que la lógica de integración ponga en riesgo la aplicación principal.
   - Conforme evolucionen los modelos de IA o surjan nuevos agentes, se puede **agregar o sustituir** componentes de forma aislada, sin migrar todo el proyecto otra vez.