Informática y comunicaciones
ies azarquiel

Red Social: “VEE”

El objetivo principal es ofrecer una alternativa a las redes sociales convencionales, proporcionando una experiencia que complemente la vida de los usuarios en lugar de consumirla.

TRABAJO FINAL DE CICLO

CFGS DESARROLLO DE APLICACIONES WEB

Autor/a: Ángel García-Page Rodríguez

Tutor/a: José Enrique Atiénzar Ibáñez

Junio de 2025

Contenido

Capítulo 1: Introducción y Objetivos

En el presente proyecto se desarrolla una aplicación web denominada VEE, una red social centrada en la publicación de diarios personales. El objetivo principal de esta plataforma es ofrecer a los usuarios un espacio digital donde puedan expresar su día a día a través de entradas escritas, acompañadas opcionalmente por imágenes, y recibir comentarios de otros usuarios que también pueden incluir contenido multimedia.
La idea surge de combinar conceptos tradicionales de blog con dinámicas sociales de plataformas actuales, fomentando una experiencia más íntima, reflexiva y centrada en la escritura y la memoria diaria.

1.1 Justificación

En un panorama dominado por redes sociales rápidas y centradas en la imagen o el vídeo, VEE busca recuperar el valor de la escritura personal como medio de expresión, combinándola con una experiencia social mínima pero significativa. La idea es crear una red en la que el texto sea protagonista, pero sin renunciar a la estética ni a las funciones básicas de interacción social.
Desde el punto de vista técnico, este proyecto permite aplicar de forma práctica los conocimientos adquiridos durante el ciclo formativo de Desarrollo de Aplicaciones Web, combinando backend, frontend y diseño de base de datos. Además, se ha buscado emplear buenas prácticas de diseño, como separación de capas, uso de DTOs, control de errores y una estructura preparada para escalar.

Capítulo 2: Especificación de Requisitos

Requisitos Funcionales

Registro de usuarios: permite crear cuentas nuevas con email, nombre y contraseña. Se valida que el email no esté repetido.

Inicio de sesión: validación mediante email y contraseña. En futuras versiones se añadirá autenticación JWT.

Publicación de diarios: cada usuario puede crear un diario al día, compuesto por un título, texto y una o varias imágenes.

Visualización de diarios: listado general o filtrado por usuario. Se prioriza mostrar los más recientes.

Comentarios: cualquier usuario puede comentar diarios ajenos, incluyendo texto y una imagen opcional.

Edición del diario propio: permite modificar contenido antes de que finalice el día.

Requisitos No Funcionales

Interfaz responsive con diseño mobile-first.

Bajo consumo de recursos y tiempos de carga mínimos.

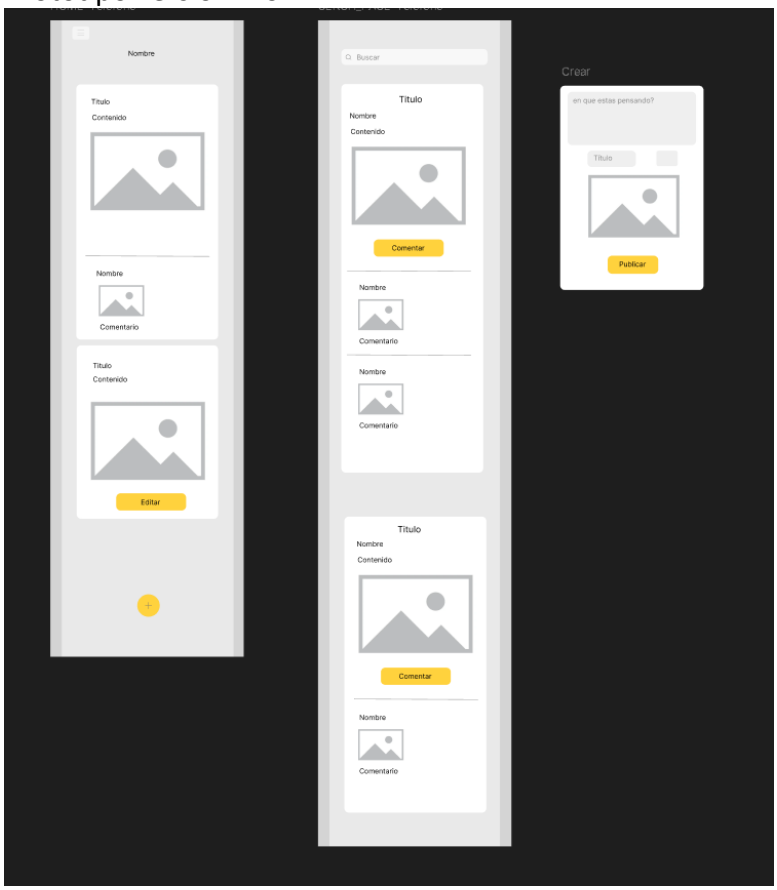
Implementación local sin despliegue obligatorio.

Seguridad básica en la validación de entrada y control de errores.

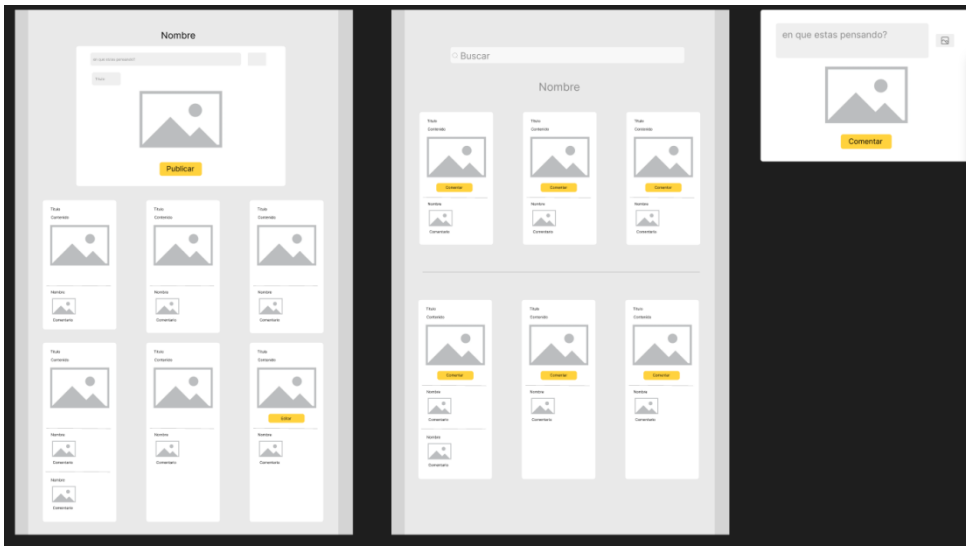
Separación estricta entre lógica de negocio, controladores y vistas.

Prototipos de la interfaz

Prototipo versión móvil:



Prototipo versión escritorio:



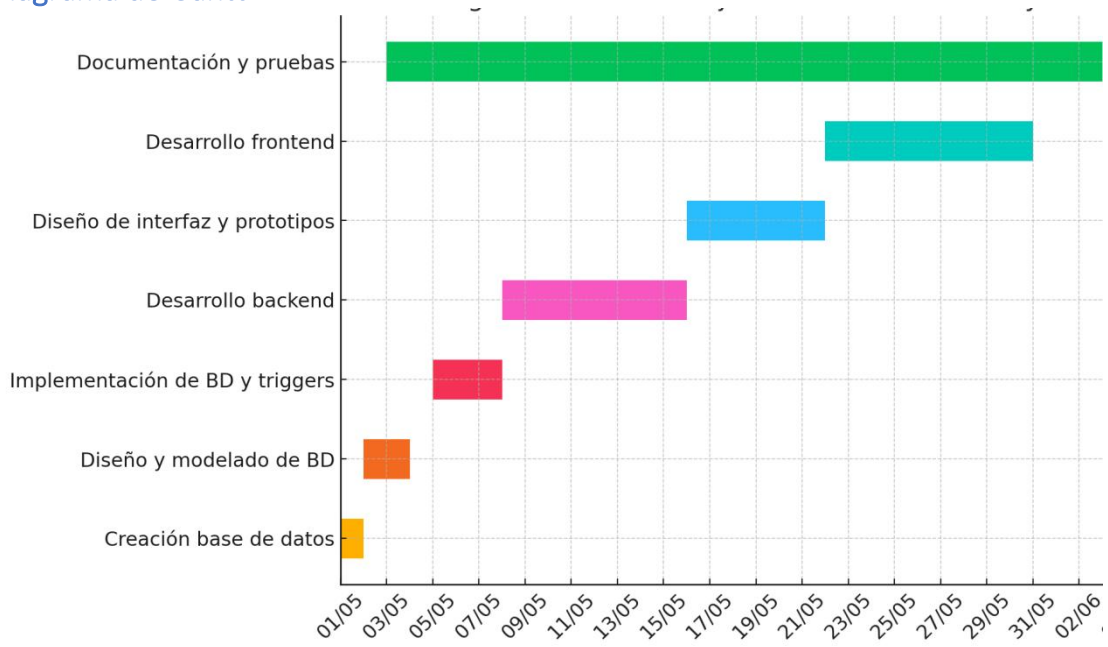
Capítulo 3: Planificación Temporal y Evaluación de Costes

La planificación del proyecto se basó en bloques de trabajo semanales, priorizando la funcionalidad sobre la optimización. Se adoptó una metodología personal tipo cascada dividida en fases: análisis, diseño, desarrollo backend, frontend, pruebas y documentación.

Resumen de fases y planificación temporal del proyecto:

- **1 de mayo:** Creación de la base de datos.
- **2-3 de mayo:** Diseño del modelo entidad-relación y normalización.
- **5-7 de mayo:** Implementación de la base de datos en Oracle, incluyendo creación de tablas, índices y triggers.
- **8-15 de mayo:** Desarrollo del backend utilizando Spring Boot y conexión a la base de datos.
- **16-21 de mayo:** Diseño de la interfaz de usuario y creación de prototipos en Figma.
- **22-30 de mayo:** Desarrollo del frontend con React y Tailwind CSS.
- **3 de mayo - 2 de junio:** Documentación del proyecto y pruebas de la aplicación.

Diagrama de Gantt



Evaluación de costes

Costes estimados del proyecto:

Concepto	Coste estimado
Hosting (uso local)	0 €
Base de datos (Oracle gratuita)	0 €
Frontend (React + Tailwind)	0 €
Diseño (Figma gratuito)	0 €
Suscripción ChatGPT Plus (1 mes)	23 €

Capítulo 4: Tecnologías Utilizadas

Backend – Spring Boot

Se utilizó Spring Boot por su arquitectura modular, facilidad de configuración y compatibilidad directa con JPA/Hibernate. Permite crear APIs RESTful de forma ágil y segura. Se aplicó inyección de dependencias, separación de capas y validaciones.

Base de Datos – Oracle

Se eligió Oracle Database por su potencia, estabilidad y recursos avanzados como triggers y secuencias. El modelo entidad-relación se implementó mediante anotaciones JPA.

Frontend – React + Tailwind CSS

React facilita la creación de componentes reutilizables, y su enfoque declarativo simplifica la gestión del estado. Tailwind CSS permite construir interfaces modernas sin necesidad de escribir CSS personalizado.

Diseño – Figma

Figma se usó para crear prototipos responsivos. Se diseñaron las vistas en formato móvil y escritorio, basadas en prácticas modernas de UX.

Herramientas de apoyo

Postman: para testear la API REST.
ChatGPT: para soporte técnico y revisión de redacción.

Capítulo 5: Desarrollo e Implementación

5.1 Estructura General del Backend

El backend de la red social VEE se ha desarrollado utilizando Spring Boot, siguiendo el patrón arquitectónico MVC (Modelo - Vista - Controlador). Se ha dividido el código fuente en paquetes lógicamente organizados:

model: contiene las entidades JPA que representan las tablas de la base de datos.

repository: interfaces que permiten realizar operaciones CRUD mediante Spring Data JPA.

dto: objetos que encapsulan datos para la comunicación con el frontend.

service y service.impl: lógica de negocio y validaciones.

controller: controlador REST que expone los endpoints de la API.

VeeApplication.java: clase principal que lanza la aplicación.

5.2 Modelo de Datos (model/)

Usuario.java

Define al usuario registrado. Campos clave: id, nombre, email, pass
Relaciones: un usuario puede tener múltiples diarios (OneToMany), comentarios e imágenes.

Diario.java

Representa la publicación diaria de un usuario. Campos: título, texto, fecha.
Relación ManyToOne con Usuario.
Relación OneToMany con Comentario e Imagen.

Comentario.java

Permite a los usuarios comentar diarios. Campos: texto, usuarioId, diarioId.
Puede tener una imagen (OneToOne opcional).

Imagen.java

Modelo reutilizable para imágenes relacionadas con comentarios o diarios.
Campos: url, título.
Relaciones ManyToOne opcionales con Comentario, Diario y Usuario.

EsAmigo.java + EsAmigoId.java

Modelo para representar relaciones de amistad entre usuarios con clave compuesta (EmbeddedId).
Campos: usuario1Id, usuario2Id, estado.
Relaciones bidireccionales mediante MapsId.

5.3 Persistencia con JPA (repository/)

Todos los repositorios extienden JpaRepository, lo que permite acceder a las entidades sin escribir SQL.

Repositorios definidos:

UsuarioRepository: búsqueda por email, nombre, ID.

DiarioRepository: diarios por usuario.

ComentarioRepository: comentarios por diario.

ImagenRepository: imágenes asociadas.

EsAmigoRepository: estado de amistad entre dos usuarios.

5.4 Transferencia de Datos (dto/)

Los DTOs se usan para encapsular la información expuesta por la API y evitar mostrar entidades directamente.

UsuarioDto: contiene id, nombre, email.

DiarioDto: id, título, texto, fecha, usuarioId, listas de imágenes y comentarios.

ComentarioDto: id, texto, usuarioId, diarioId, imagen asociada.

ImagenDto: id, url, título, relaciones a diario o comentario.

Esto facilita la validación, protege el modelo de datos y mejora la comunicación con React.

5.5 Lógica de Negocio (service/ y service.impl/)

Cada entidad cuenta con una interfaz Service y una clase ServiceImpl que encapsulan la lógica de negocio del sistema. Estas clases actúan como capa intermedia entre los controladores (API REST) y los repositorios (acceso a base de datos). Su función principal es aplicar reglas de validación, garantizar la integridad de los datos y preparar respuestas adecuadas (DTOs).

UsuarioServiceImpl

1. *List<UsuarioDto> findAll()*

Devuelve una lista de todos los usuarios registrados en el sistema. Se obtienen las entidades desde la base de datos y se transforman en DTOs.

2. *UsuarioDto findById(Long id)*

Busca un usuario por su ID. Si existe, devuelve su representación como UsuarioDto. Si no, lanza una excepción o retorna null según la implementación.

3. *List<UsuarioDto> findByUsername(String username)*

Devuelve todos los usuarios cuyo nombre coincida (o contenga) el string proporcionado. Útil para búsquedas o filtrado.

4. *void save(UsuarioDto usuarioDto)*

Guarda un nuevo usuario a partir del DTO recibido. Convierte el DTO en entidad Usuario, lo valida y lo persiste usando usuarioRepository.

5. *void save(UsuarioDto usuarioDto, Long id)*

Versión sobrecargada del método anterior. Permite actualizar los datos de un usuario ya existente identificado por su id.

6. *void delete(Long id)*

Elimina el usuario cuyo ID coincide. Antes de borrar, se podría validar si existen relaciones (diarios, comentarios…).

DiarioServiceImpl

1.List<DiarioDto> findAll()

Devuelve todos los diarios de la plataforma. Transforma cada entidad Diario en un DiarioDto.

2.void save(DiarioDto diarioDto)

Guarda un nuevo diario con los datos recibidos desde el frontend. Asocia el diario a un usuario si el ID es válido.

3. void save(DiarioDto diarioDto, Long id)

Actualiza un diario ya existente. Usa el id para localizarlo y luego aplica los cambios desde el DTO.

4.void delete(Long id)

Elimina un diario por su ID. Se encarga de borrar también sus comentarios e imágenes asociadas si procede.

ComentarioServiceImpl

1.List<ComentarioDto> findAll()

Recupera todos los comentarios del sistema. Convierte cada entidad Comentario en su correspondiente DTO.

2.void save(ComentarioDto comentarioDto)

Crea un nuevo comentario a partir del DTO.

Valida que el usuario y el diario existen.

Si hay imagen adjunta, la guarda y la enlaza.

3. void save(ComentarioDto comentarioDto, Long id)
Permite actualizar un comentario existente. Se localiza por id y se actualizan sus campos.

4. *void deleteById(Long id)*

Elimina un comentario específico usando su identificador.

ImagenServiceImpl

1. List<ImagenDto> findAll()

Devuelve todas las imágenes de la base de datos, transformadas en ImagenDto.
Incluye imágenes asociadas a diarios o comentarios.

2. void save(ImagenDto imagenDto)

Guarda una nueva imagen en la base de datos.

Puede asociarse a un diario o un comentario, según los IDs que lleguen.

Se valida que las entidades asociadas existan.

3. void delete(Long id)

Elimina una imagen por su ID.

5.6 Controlador REST (controller/)

El controlador principal de la aplicación es VeeController.java. Esta clase está anotada con @RestController y expone todos los endpoints REST utilizados por el frontend para interactuar con el backend. Implementa operaciones CRUD completas (crear, leer, actualizar y borrar) para las entidades Usuario, Diario, Comentario e Imagen.

Endpoints definidos

Usuarios

- `GET /usuarios`
  Devuelve la lista completa de usuarios registrados.
  → `List<UsuarioDto> usuarios()`
- `GET /usuarios/{id}`
  Devuelve los datos de un usuario específico según su ID.
  → `UsuarioDto usuario(Long id)`
- `GET /usuarios/{username}`
  Devuelve todos los usuarios cuyo nombre coincide con el `username`.
  → `List<UsuarioDto> usuarioByUsername(String username)`
- `POST /usuarios`
  Registra un nuevo usuario en el sistema.
  → `void saveUsuario(UsuarioDto usuarioDto)`

- `PUT /usuarios/{id}`
  Actualiza los datos de un usuario existente.
  → `void updateUsuario(Long id, UsuarioDto usuarioDto)`
- `DELETE /usuarios/{id}`
  Elimina un usuario por su ID.
  → `void deleteUsuario(Long id)`

Diarios

- `GET /diarios`
  Devuelve todos los diarios disponibles en la plataforma.
  → `List<DiarioDto> diarios()`
- `POST /diarios`
  Crea un nuevo diario.
  → `void saveDiario(DiarioDto diarioDto)`
- `PUT /diarios/{id}`
  Actualiza un diario existente.
  → `void updateDiario(Long id, DiarioDto diarioDto)`
- `DELETE /diarios/{id}`
  Elimina un diario por su identificador.
  → `void deleteDiario(Long id)`

Comentarios

- `GET /comentarios`
  Devuelve todos los comentarios registrados.
  → `List<ComentarioDto> comentarios()`
- `POST /comentarios`
  Crea un nuevo comentario vinculado a un diario.
  → `void saveComentario(ComentarioDto comentarioDto)`
- `PUT /comentarios/{id}`
  Modifica un comentario ya existente.
  → `void updateComentario(Long id, ComentarioDto comentarioDto)`
- `DELETE /comentarios/{id}`
  Elimina un comentario por su ID.
  → `void deleteComentario(Long id)`

Imágenes

- `GET /imagenes`
  Devuelve todas las imágenes almacenadas en el sistema.
  → `List<ImagenDto> imagenes()`
- `POST /imagenes`
  Sube una nueva imagen asociada a un diario o comentario.
  → `void saveImagen(ImagenDto imagenDto)`
- `DELETE /imagenes/{id}`
  Elimina una imagen por su identificador.
  → `void deleteImagen(Long id)`

Notas técnicas

- Todos los métodos usan `@RequestBody` para recibir DTOs del frontend y `@PathVariable` para parámetros de ruta.
- Las operaciones `GET` devuelven listas o elementos individuales, mientras que `POST`, `PUT` y `DELETE` son void.
- La lógica interna está delegada completamente a los servicios correspondientes, respetando el patrón de capas.

5.7 Aplicación Principal (VeeApplication.java)

Contiene el punto de entrada de la aplicación:

```
@SpringBootApplication
public class VeeApplication {
public static void main(String[] args) {
SpringApplication.run(VeeApplication.class, args);
}
}
```

Esto configura automáticamente los beans, la base de datos, controladores y servicios.

5.8 Pruebas y Validaciones

Durante el desarrollo se utilizaron herramientas como Postman para probar los endpoints manualmente. Se insertaron datos de prueba directamente en Oracle, y se validaron:

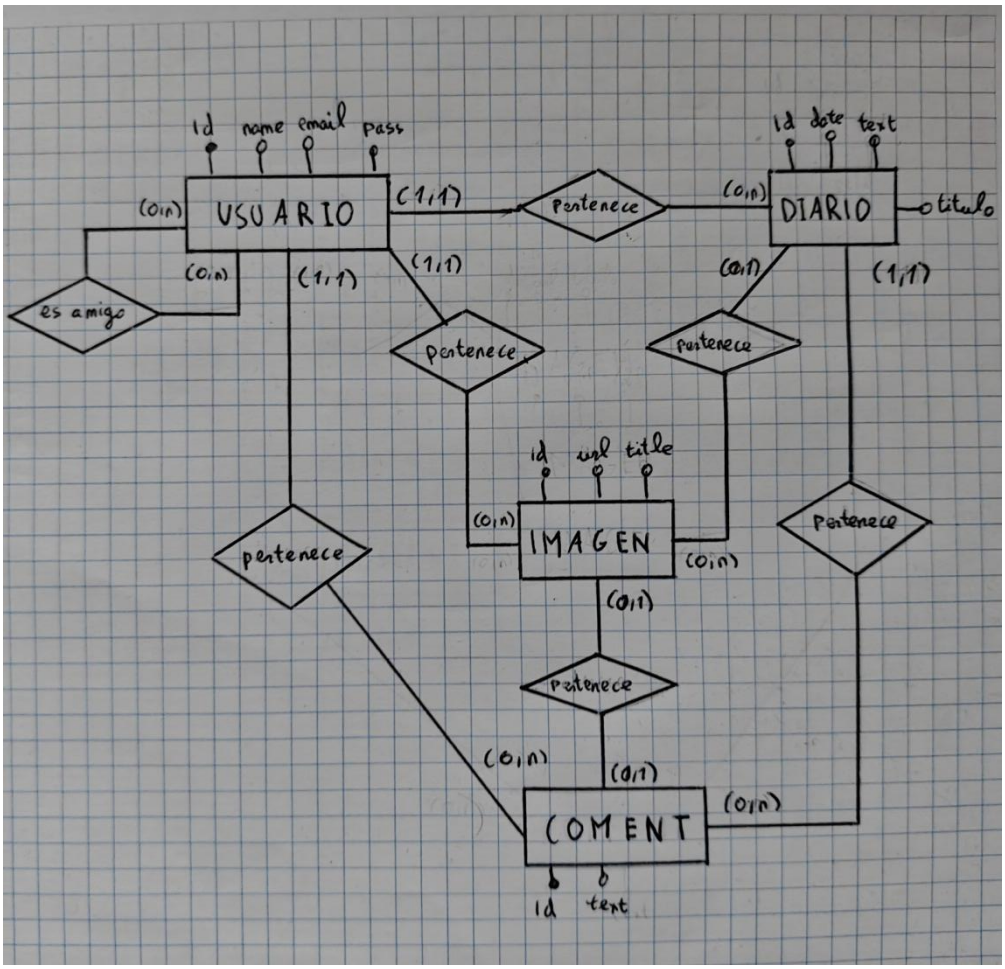
Inserciones de usuarios, diarios y comentarios.

Correcta relación entre entidades.

Serialización y estructura de respuesta de los DTOs.

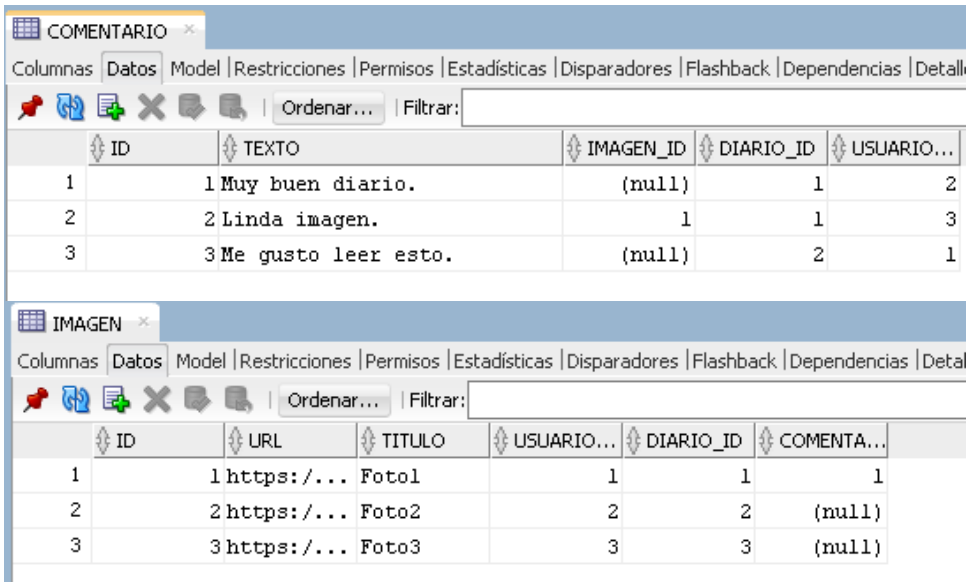
Validaciones de existencia antes de guardar o vincular datos.

Las siguientes capturas muestran el modelo entidad-relación diseñado, incluyendo la definición de relaciones entre entidades como 'Usuario', 'Diario', 'Comentario' e 'Imagen'.



En el modelo se observa que cada 'Usuario' puede tener múltiples 'Diarios', y cada 'Diario' múltiples 'Comentarios' e 'Imágenes'. Los 'Comentarios' a su vez pueden incluir imágenes, y todos los elementos están correctamente enlazados mediante claves foráneas.

Además, se incluyen capturas de las tablas generadas con datos de prueba. Esto evidencia que la estructura diseñada es funcional, y que el sistema de inserción mediante triggers funciona correctamente. Los IDs se autogeneran utilizando secuencias, lo cual mejora la integridad y eficiencia del sistema.



COMENTARIO

Columnas | Datos | Model | Restricciones | Permisos | Estadísticas | Disparadores | Flashback | Dependencias | Detall

Ordenar... | Filtrar:

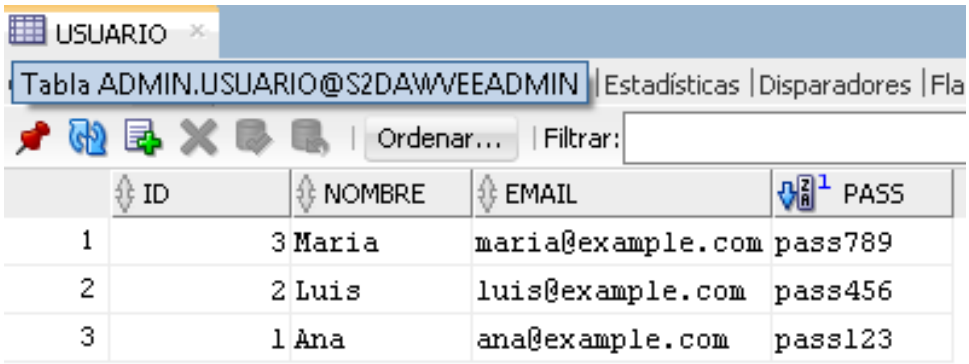
	ID	TEXTO	IMAGEN_ID	DIARIO_ID	USUARIO...
1	1	Muy buen diario.	(null)	1	2
2	2	Linda imagen.	1	1	3
3	3	Me gusto leer esto.	(null)	2	1

IMAGEN

Columnas | Datos | Model | Restricciones | Permisos | Estadísticas | Disparadores | Flashback | Dependencias | Detal

Ordenar... | Filtrar:

	ID	URL	TITULO	USUARIO...	DIARIO_ID	COMENTA...
1	1	https:/...	Foto1	1	1	1
2	2	https:/...	Foto2	2	2	(null)
3	3	https:/...	Foto3	3	3	(null)

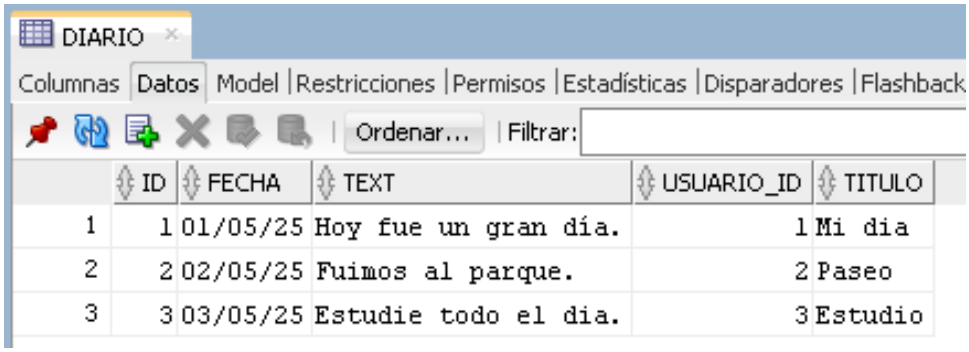


USUARIO

Tabla ADMIN.USUARIO@S2DAW\VEEADMIN | Estadísticas | Disparadores | Fla

Ordenar... | Filtrar:

	ID	NOMBRE	EMAIL	PASS
1	3	Maria	maria@example.com	pass789
2	2	Luis	luis@example.com	pass456
3	1	Ana	ana@example.com	pass123



DIARIO

Columnas | Datos | Model | Restricciones | Permisos | Estadísticas | Disparadores | Flashback

Ordenar... | Filtrar:

	ID	FECHA	TEXT	USUARIO_ID	TITULO
1	1	01/05/25	Hoy fue un gran día.	1	Mi dia
2	2	02/05/25	Fuimos al parque.	2	Paseo
3	3	03/05/25	Estudie todo el dia.	3	Estudio

5.9 Razón por la que no se implementó la funcionalidad de solicitudes de amistad

Durante el desarrollo del backend, se diseñó una entidad llamada `EsAmigo` para modelar las relaciones de amistad entre usuarios. Esta entidad permite, mediante una clave compuesta (`usuario1Id` y `usuario2Id`), representar relaciones bidireccionales y almacenar un posible estado de la relación (pendiente, aceptada, rechazada, etc.).
Sin embargo, **la funcionalidad completa de solicitudes de amistad no se llegó a implementar** por las siguientes razones:
Limitación de tiempo del proyecto: El desarrollo de un sistema de amistad con lógica de solicitudes, aceptaciones, notificaciones y comprobaciones bidireccionales requiere una lógica adicional compleja que no era prioritaria frente a otras partes más esenciales del sistema, como los diarios, los comentarios o la subida de imágenes.
Dificultad añadida en la validación de operaciones: La implementación de esta función habría requerido mecanismos adicionales para validar acciones según el estado de amistad entre usuarios (por ejemplo, restringir ver diarios solo si hay amistad aceptada), lo cual habría incrementado significativamente la complejidad del backend y del frontend.
A pesar de no haberse incluido en esta versión, **la estructura de datos y la entidad `EsAmigo` ya están preparadas para una futura expansión del sistema**, por lo que la funcionalidad de amistad podría incorporarse fácilmente más adelante con control de estados, endpoints REST dedicados y validaciones de acceso basadas en la relación entre usuarios.

5.10 Desarrollo del Frontend

El frontend de VEE fue desarrollado utilizando **React 18** y **TypeScript**, siguiendo una estructura modular y un enfoque mobile-first, complementado con **Tailwind CSS** para la gestión de estilos. Esto permitió crear una interfaz moderna, ligera y fácil de mantener. La comunicación con el backend se realiza mediante llamadas a la API REST implementada con Spring Boot, usando tanto fetch como axios para gestionar las peticiones HTTP.
La interfaz está compuesta por distintas páginas que representan las vistas de usuario, y cada una está subdividida en componentes reutilizables. Además, se estructuró el proyecto con una clara separación entre la lógica de presentación, gestión de estado y servicios de conexión a la API.

Estructura del proyecto

La arquitectura del frontend está dividida en las siguientes carpetas:
components/: componentes reutilizables como DiarioCard, ComentarioCard, EditorDiario, etc.
pages/: vistas principales (Home, Search, Welcome).
services/: funciones que interactúan con la API (GET, POST, PUT, DELETE).
App.tsx: punto de entrada principal que contiene la lógica de enrutamiento.
index.tsx: archivo base donde se monta la aplicación.

Descripción detallada de las páginas

Welcome.tsx (Página de bienvenida)

Esta página presenta dos formularios: uno para registrarse y otro para iniciar sesión. Utiliza hooks como useState y useEffect para manejar los datos del formulario. Las credenciales del usuario se envían a la API (POST /usuarios) para registrar un nuevo usuario o iniciar sesión.
Errores actuales:

- El login no mantiene la sesión tras recargar.
- Aún no se ha implementado autenticación con JWT ni almacenamiento en localStorage.

Home.tsx (Página principal del usuario)

Es la vista principal tras el inicio de sesión. Contiene:

- El componente EditorDiario, visible permanentemente en escritorio.
- Una lista de diarios ya creados por el usuario.

El contenido del diario es enriquecido gracias a Quill y puede incluir imágenes.

Search.tsx (Página de exploración de usuarios)

En esta vista se muestran todos los usuarios de la base de datos y sus respectivos diarios. Permite buscar usuarios por nombre mediante una barra de búsqueda que lanza peticiones GET /usuarios/{nombre}.
Cada resultado muestra:

- El nombre del usuario.
- Todos los diarios públicos asociados a ese usuario.
- Comentarios e imágenes asociadas.

Editor Quill

Se utilizó react-quill para la creación y edición de diarios. El editor es un componente WYSIWYG que genera contenido HTML enriquecido.

Características destacadas:

- Barra de herramientas con opciones como encabezados, listas, texto en negrita, cursiva, enlaces e imágenes.
- Posibilidad de pegar o subir imágenes que se convierten a base64 y luego se envían al backend.
- El contenido HTML se guarda en la base de datos y se renderiza directamente en el componente DiarioCard.

Limitaciones:

- Advertencias por eventos DOMNodeInserted (obsoletos).
- Falta una vista previa para las imágenes.
- En móviles aún no se implementa la edición como modal flotante.

Componentes del sistema

DiarioCard

Este componente se encarga de representar visualmente un diario completo. Sus funciones principales son:
Mostrar el **título del diario**, su **fecha**, y el **contenido** en formato HTML (renderizado desde el resultado de Quill).
Mostrar todas las **imágenes** asociadas a ese diario.
Listar los **comentarios** relacionados.
Incluir un botón “Editar” cuando el diario pertenece al usuario autenticado.

Props principales:

- diario: objeto DiarioDto con título, texto, lista de imágenes y comentarios.
- editable: booleano que determina si se debe mostrar el botón de edición.

Lógica destacada:

- Se usa dangerouslySetInnerHTML para renderizar el HTML generado por Quill.
- Se mapea la lista de imágenes para mostrarlas con .
- Se renderizan componentes ComentarioCard para cada comentario.

ComentarioCard

Este componente representa un único comentario realizado sobre un diario.

Funciones principales:

- Mostrar el **autor** (nombre del usuario), el **texto del comentario** y una **imagen**, si está presente.
- Visualmente es una tarjeta pequeña que aparece debajo de un diario.

Props principales:

- comentario: objeto ComentarioDto con texto, imagen, autor y diarioId.

Lógica destacada:

- Si `comentario.imagen` existe, se muestra en miniatura.
- Se muestra el nombre del autor del comentario mediante un acceso a `usuario.nombre` si está presente.

Pendiente:

- Implementar botones para editar y eliminar el comentario.

EditorDiario

Es el componente más complejo del sistema. Permite crear o editar diarios usando el editor Quill.

Funciones principales:

- Recibir un **diario existente** para editar o funcionar vacío para crear uno nuevo.
- Permitir escribir un **título** y un **texto** enriquecido con formato HTML.
- Capturar el contenido y enviarlo a la API mediante POST o PUT según el caso.

Hooks usados:

- useState para gestionar título, contenido y modo edición.
- useEffect para rellenar los campos si se trata de una edición.
- useRef para referenciar el editor de Quill.

Lógica destacada:

- Se configura Quill con un modules que define la barra de herramientas (bold, headers, links, images...)
- El contenido del editor se obtiene mediante editorRef.current.getEditor().getHTML().
- Enviar los datos como DiarioDto con usuarioId, titulo, texto al backend.

Validaciones:

- No se permite guardar si el campo de título está vacío.
- Falta mostrar mensajes de éxito o error.

Futuro:

- Añadir un input para subir imágenes arrastrando.
- Implementar modal en móviles.

UsuarioCard.tsx

Componente exclusivo de la vista Search. Muestra información sobre un usuario encontrado y sus diarios.

Funciones principales:

- Mostrar el **nombre del usuario**.
- Mostrar cada uno de sus **diarios** utilizando DiarioCard.

Props:

- usuario: objeto UsuarioDto que contiene diarios: DiarioDto[].

Lógica:

- Usa usuario.diarios.map(...) para renderizar múltiples diarios.
- Cada DiarioCard se renderiza como parte del mismo bloque visual.

Con estos componentes bien separados, se logró una interfaz intuitiva, mantenible y orientada a la reutilización. Las props fueron tipadas con TypeScript para evitar errores y facilitar la integración con los datos del backend.

Llamadas a la API

Las llamadas se realizan usando funciones personalizadas en services/api.ts, utilizando fetch o axios.

Principales operaciones:

- **GET /usuarios**: obtiene todos los usuarios.
- **GET /usuarios/{nombre}**: filtra usuarios por nombre.
- **GET /diarios**: obtiene todos los diarios del sistema.
- **POST /diarios**: crea un nuevo diario.
- **PUT /diarios/{id}**: actualiza un diario.
- **POST /comentarios**: crea un nuevo comentario.
- **POST /imagenes**: sube una imagen (diario o comentario).

Cada función transforma la respuesta en JSON y la gestiona según el contexto del componente. Se hace uso extensivo de useEffect para cargar datos al montar los componentes.

Gestión del estado

En la gestión de diarios con Quill, el estado cumple una doble función: controlar el formulario y reflejar en todo momento el contenido que se va a enviar al backend.

Estado en la creación de un diario

Cuando un usuario quiere crear un nuevo diario, el componente `EditorDiario` se monta con los campos vacíos. Los hooks principales son:

```
const [titulo, setTitulo] = useState("");
const [contenido, setContenido] = useState("");
```

- setTitulo se ejecuta en el campo de entrada del título.
- setContenido se sincroniza con el contenido del editor Quill, escuchando los cambios mediante el evento onChange.

El editor de Quill genera internamente un contenido en formato HTML. Cada vez que el usuario escribe o edita, se actualiza el estado:

```
<ReactQuill value={contenido} onChange={setContenido} />
```

Al pulsar en “Guardar”, se construye un objeto tipo DiarioDto:

```
const nuevoDiario = {
  titulo: titulo,
  texto: contenido,
  usuarioId: idUsuarioActual
};
```

Este objeto se envía mediante fetch o axios al backend.

Estado en la edición de un diario

En modo edición, el componente recibe como props un objeto diario ya existente. Al montar el componente:

```
useEffect(() => {
  if (diario) {
    setTitulo(diario.titulo);
    setContenido(diario.texto);
  }
}, [diario]);
```

Esto asegura que el editor Quill y el campo de título se inicialicen con los datos previamente guardados. El usuario puede modificar el texto o las imágenes, y al pulsar "Guardar", se ejecuta una función que hace PUT /diarios/{id}.

Diferencias clave:

- En creación, el estado parte de valores vacíos.
- En edición, el estado se inicializa con datos recibidos por props.
- En ambos casos, el estado mantiene sincronizado el contenido del editor con la interfaz.

Importancia de useState:

- Permite reflejar cambios inmediatos al usuario.
- Evita inconsistencias entre lo que se ve y lo que se envía.
- Facilita validaciones antes de guardar (ej. título vacío).

Relación directa con Quill:

- Cada cambio en el editor actualiza contenido.
- El valor de contenido es el que se guarda como texto HTML en la base de datos.
- Así se logra persistencia exacta del contenido con formato (negrita, listas, enlaces, imágenes, etc.).

Diseño Responsive

Gracias a Tailwind CSS, la aplicación se adapta automáticamente a móviles, tablets y pantallas de escritorio. El diseño mobile-first prioriza la usabilidad en dispositivos pequeños, especialmente en la navegación y el uso del editor.

En escritorio:

- El editor está siempre visible.
- Se muestran múltiples diarios en cuadrícula.

En móvil:

- El editor se lanzará como una modal flotante (pendiente de implementación).
- Los diarios se listan en una sola columna.

Problemas técnicos detectados

- La autenticación aún no se conserva entre sesiones.
- Falta protección de rutas (login requerido).
- Sin paginación para resultados largos.
- Advertencias técnicas en consola por eventos DOM obsoletos.
- No se ha implementado subida múltiple de imágenes ni edición de comentarios....

Capítulo 6: Conclusiones y Líneas Futuras

Valoración general del proyecto

La realización de este proyecto ha supuesto un desafío completo, tanto en lo técnico como en lo personal. Se han puesto en práctica múltiples conocimientos adquiridos durante el ciclo de Desarrollo de Aplicaciones Web (DAW), enfrentándose a un entorno realista de construcción de una aplicación completa. La planificación inicial tuvo que adaptarse progresivamente para centrarse en las funcionalidades esenciales, sacrificando algunas ideas originales para asegurar una entrega funcional, mantenible y coherente.

Comparación con los objetivos iniciales

En el planteamiento inicial se pretendía construir una red social con:

- Sistema de registro e inicio de sesión funcional y seguro.
- Creación y edición de diarios enriquecidos.
- Gestión de imágenes en publicaciones y comentarios.
- Comentarios entre usuarios.
- Sistema de amistad con solicitudes, confirmaciones y restricciones de visibilidad.

- Autenticación persistente mediante tokens JWT.
- Despliegue completo del sistema en servicios gratuitos online.

Aunque algunas de estas metas se cumplieron parcialmente o fueron implementadas de forma básica, otras debieron posponerse debido a la complejidad y al límite temporal del proyecto.

Logros destacados

- Backend completo y funcional: Se logró desarrollar una API REST estructurada en capas, con validaciones robustas mediante DTOs y lógica de negocio clara, conectada a Oracle Cloud.
- Frontend moderno y responsive: React 18 con TypeScript y Tailwind CSS permitió crear una interfaz visual limpia y adaptable tanto en móvil como escritorio.
- Editor enriquecido: La integración de React Quill facilitó la creación de contenido con formato HTML, incluyendo imágenes.
- Gestión de imágenes y comentarios: Se logró conectar correctamente imágenes a diarios y comentarios, ampliando la expresividad del contenido.
- Componentes reutilizables: La aplicación está estructurada con componentes que permiten escalar o modificar funcionalidades sin romper el resto del sistema.

Dificultades técnicas superadas

- Gestión del estado en React: La diferencia entre crear y editar diarios exigió un control preciso del estado del formulario y sincronización con el editor Quill.
- Comunicación con la API: Fue necesario tipar y validar cada llamada, manejando correctamente respuestas, errores y datos anidados.
- Integración del editor Quill: Supuso un reto técnico tanto por su configuración como por su interacción con la estructura del backend.
- Responsividad: Conseguir que la interfaz se adaptara a diferentes tamaños de pantalla fue más complejo de lo previsto, especialmente en los formularios y el editor.

Motivos por los que no se desarrollaron algunas funciones

- Sistema de amistad: Aunque la entidad EsAmigo se encuentra implementada, su lógica era demasiado extensa y no esencial para la funcionalidad básica, por lo que se optó por dejarlo preparado para una futura expansión.
- Autenticación JWT: Se consideró implementar JWT para gestionar sesiones, pero por cuestiones de tiempo y estabilidad se priorizó una experiencia funcional con registro e inicio de sesión básico.
- Persistencia de sesión: El frontend aún no guarda el estado del usuario tras iniciar sesión, lo que impide navegar entre rutas protegidas. Esta funcionalidad está planificada pero no se logró integrar.
- Modal para móviles y confirmaciones visuales: Por falta de tiempo, no se implementaron retroalimentaciones visuales (como toasts) ni la versión modal del editor Quill en dispositivos móviles.

Conclusión final

El sistema desarrollado, a pesar de no ser completo en todas sus aspiraciones iniciales, representa una base funcional y sólida para continuar evolucionando. Las funcionalidades centrales están implementadas, probadas y organizadas de forma clara, lo que permitirá futuras mejoras con poco esfuerzo de refactorización.
La decisión de recortar el alcance en ciertas áreas y centrar los esfuerzos en un producto estable fue adecuada y justificada, priorizando la calidad y coherencia sobre la cantidad. Este proyecto ha servido para consolidar habilidades prácticas de desarrollo web completo (full stack), desde el diseño hasta la implementación técnica.

Capítulo 7: Bibliografía

- Fernando Herrera (Udemy). React: De cero a experto (Hooks y MERN)
- Documentación de Spring Boot
- Documentación de React
- Oracle Database Documentation
- Tailwind CSS Docs
- Figma Learn
- ChatGPT (OpenAI): asistencia técnica durante el desarrollo.
- Materiales del profesor Julio Efrén: teoría de bases de datos y JPA.