

# Trabajo de Fin de Grado

---

Miguel Sánchez Vázquez

Eduardo Díaz Sánchez

Javier Arnedo Torres

# Índice

1. Propósito de nuestro proyecto
2. Tecnologías utilizadas
3. Estructura del proyecto
4. Funcionalidades Principales
5. Implementación de Funcionalidades.
6. Despliegue y configuración
7. Conclusiones y futuras mejoras.



## **1. Propósito de nuestro proyecto**

# ¿Por qué este proyecto?

La gestión de guardias en centros educativos es un proceso crítico pero que consume demasiados recursos administrativos cuando se realiza manualmente.

## **Problemas que resolvemos:**

- Procesos manuales lentos y propensos a errores
- Falta de coordinación entre profesorado y dirección
- Dificultad para distribuir equitativamente las guardias
- Ausencia de datos para analizar y mejorar el proceso

## **Nuestra solución:**

Hemos desarrollado una aplicación web que digitaliza y optimiza la gestión de guardias en centros educativos, abarcando desde el registro de ausencias hasta la creación, asignación y seguimiento de las mismas.



## 2. Tecnologías utilizadas

# La elección del lenguaje de programación

Antes de llegar a una decisión, realizamos un estudio previo de las diferentes tecnologías que estaban a nuestra disposición.

Llegamos a la conclusión de unos requisitos diferentes que tenían que cumplir :

- La tecnología ha de ser capaz de crear un frontend.
- Tenemos que ser capaces de entender algo de la tecnología.
- Preferiblemente, algo que ya hayamos visto, para no ir tan apretados de tiempo.

# Tecnologías posibles para el desarrollo

Tras explorar las opciones, llegamos a la conclusión de unos candidatos los cuales eran óptimos para desarrollar nuestro proyecto.

- Java.
- Web.
- Flutterflow.

# Java

## Pros:

- Tenemos experiencia en java y las funciones podrían realizarse sin mucho problema.
- Se podría crear un backend customizado si es necesario enteramente en java.

## Cons:

- La única opción gráfica que ofrece java es : JAVA FX. La cual no solo se ve anticuada, sino que no tenemos experiencia previa y es un problema muy grande teniendo tiempo limitado.



# FlutterFlow

## Pros:

- Capacidad de maquetar un front de una forma bastante buena.
- No tiene mucha dificultad a la hora de trabajar con la herramienta.

## Cons:

- No permite trabajar de una forma cómoda con github.
- No tiene una forma buena de comentar el código fuente salvo que se dejen comentarios en los componentes.

# Desarrollo web con React, Angular

Elegimos desarrollar la aplicación en web con React por las siguientes razones:

- Tenemos una forma clara de tener un control de versiones en git.
- Podemos desarrollar un frontend de calidad.
- Tenemos algo de experiencia en el desarrollo web.
- Tiene integración con supabase, el cual usaremos cómo backend.

# Desarrollo web con React, Angular

Las tecnologías elegidas para el proyecto por tanto son:

- Next.js : Framework de React que ofrece funcionalidades como renderizado del lado del servidor y generación de sitios estáticos
- Type Script : Superset de JavaScript que añade tipado estático, mejorando la estabilidad y mantenibilidad del código.
- Bootstrap : Framework de CSS que facilita el diseño responsive y una interfaz de usuario coherente.




### 3. Estructura del proyecto

# Carpeta Components

Esta carpeta, es encontrada en los proyectos de tanto Angular, Vue, o React, aquí es dónde están contenidos todos los componentes reutilizables de nuestro proyecto.

En nuestro caso, iremos a este componente de la interfaz llamado “avatar.tsx”

 avatar.tsx



El formato tsx, es typescript, el cual utiliza el lenguaje JSX, el cual es muy propio de React.

# Carpeta Hooks

La carpeta hooks, es la que tiene los hooks personalizados de React.

Los hooks encapsulan la lógica de negocio reutilizable de los React Hooks y pueden estar hechos tanto en Javascript, cómo en JSX.

En nuestro caso, dos hooks personalizados serian :

 use-form.ts use-mobile.tsx

# Otras carpetas misceláneas

- **src :**

Esta es la carpeta base del proyecto, dónde están tanto los componentes, hooks y etc...

- Es interesante de que todos los archivos que estén dentro de src, vayan a ser procesados mediante webpack.

- **public :**

Estos archivos son estáticos y a diferencia de src, no van a pasar por webpack, por lo que van a ser mostrados cómo tal en la aplicación.

# Otras carpetas misceláneas

- **lib :**

Aquí es dónde se van a guardar tanto las funciones reutilizables y los helpers.

- Los helpers son funciones auxiliares las cuales pueden ser reutilizadas en cualquier sitio y además, ayudan bastante a mantener el código limpio.

- **scripts :**

Esta carpeta es dónde irán a parar los scripts personalizados para realizar funciones concretas dentro del código.





## 4. Funcionalidades Principales

# Gestión y Registro de Ausencias

Los profesores pueden notificar sus ausencias de forma rápida y sencilla a través del sistema.

Se genera un registro centralizado donde se almacenan los datos de la ausencia, incluyendo el motivo , duración (hora de clase que falte ) y tarea si el profesor desea asignar.

Facilita la organización y permite una gestión más eficiente tanto a los profesores como a los administradores

# Métodos de Asignación

El sistema solo nos permite asignar guardias de manera manual es decir:

El profesor puede asignarse a una guardia pendiente, siempre y cuando se lo permita su horario de guardias/disponibilidad.

O bien, el administrador puede asignar a un profesor (con la misma condición, solo si su horario se lo permite).

Si lo asigna el administrador , saldrá un mensaje de que se ha guardado correctamente y cuando ese profesor se meta al sistema le saldrá una guardia que le han asignado pendiente de firmar.

# Monitorización en Tiempo Real

Panel de Control :

El administrador podrá ver en su panel de control , las guardias asignadas , las pendientes y las ausencias que ocurran en tiempo real.

El profesor podrá ver en su panel de control las guardias Pendientes que quedan en sistema , las que él tenga asignadas , las guardias que haya firmado de hacer y las horas de guardia totales que tiene en esa misma semana

La principal ventaja es que nos facilita a la hora de cualquier imprevisto de cualquier profesor y así pueden estar todos preparados y avisados solo con echar un vistazo a la aplicación.

```
8  
9 require 'capybara/rspec'  
10 require 'capybara/re rails'  
11  
12 Capybara.javascript_driver = :webkit  
13 Category.delete_all; Category.create  
14 Shoulda::Matchers.configure do |config|  
15   config.integrate do |integrate|  
16     with.test_framework :rspec  
17     with.library :rails  
18   end  
19 end
```

```
20 # Add additional requires below this line. This will be added to the spec file.  
21  
22 # Requires supporting ruby files with support for Capybara, e.g. Capybara::RSpecHelpers.  
23 # spec/support/ and its subdirectories. These files will be loaded by default.  
24 # run as spec files by default. This will be required by default.  
25 # in _spec.rb. It is not required by default.  
26
```

## 5. Implementación de funcionalidades

# Arquitectura general

## Organización en Capas

Nuestro sistema sigue una arquitectura de capas claramente definida que separa responsabilidades y facilita el mantenimiento:

- **Presentación:**
  - **Componentes** React y páginas Next.js organizadas por roles
  - Interfaz responsiva con Bootstrap
- **Lógica de Negocio:**
  - **Contextos** de React (GuardiasContext, AusenciasContext)
  - Gestión centralizada del estado de la aplicación
  - Validaciones de reglas de negocio específicas de los requisitos funcionales
- **Acceso a Datos:**
  - **Servicios** especializados para cada entidad (guardiasService, ausenciasService) en lib/
  - Comunicación directa con Supabase
  - Mapeo entre modelos DB y objetos de la aplicación

# Arquitectura general

Hemos intentado que la implementación tuviera un paralelismo interesante con la arquitectura tradicional de aplicaciones Java EE que aprendimos en la asignatura de Acceso a datos, aunque adaptada al paradigma de React y TypeScript.

Mientras que en Java EE, por ejemplo, se utilizaría un patrón DAO (Data Access Object) para separar la lógica de acceso a datos, en nuestra aplicación web moderna hemos implementado un enfoque similar con la siguiente estructura basado en **contextos** y **servicios**.

# Capa de acceso a datos: Servicios

En la carpeta *lib/* encontramos servicios como *guardiasService.ts* o *ausenciasService.ts* que encapsulan todas las operaciones de acceso a datos. Estos servicios son el único punto de contacto con **Supabase**, similar a cómo los **DAOs** en Java serían el único punto de contacto con **JDBC**.

```
1 // Fragmento de guardiasService.ts
2 export async function getGuardiaById(id: number): Promise<GuardiaDB | null> {
3   try {
4     const { data, error } = await supabase
5       .from('guardias')
6       .select('*')
7       .eq('id', id)
8       .single();
9
10    if (error) throw error;
11    return data;
12  } catch (error) {
13    console.error('Error al obtener guardia:', error);
14    return null;
15  }
16 }
```



## Capa de lógica de negocio: Contextos

En lugar de utilizar **clases de servicio** que agrupan la lógica de negocio, como es habitual en Java, en nuestra aplicación de React empleamos **Contextos** para gestionar y proveer funcionalidades a nivel global. Estos contextos actúan de forma similar a los controladores del modelo **MVC**, ya que encapsulan la lógica de negocio y el manejo del estado, facilitando el acceso a dicha funcionalidad desde cualquier componente de la aplicación. Por ejemplo, el ***GuardiasContext*** centraliza toda la lógica relacionada con la gestión de guardias y la expone para que cualquier componente pueda utilizarla. Todos estos contextos se encuentran organizados en la carpeta ***src/contexts***.

# Capa de lógica de negocio: Contextos

```
1 // Fragmento de GuardiasContext.tsx
2 export const GuardiasProvider = ({ children }) => {
3   const [guardias, setGuardias] = useState<Guardia[]>([]);
4
5   // Función equivalente a un método de un Service en Java
6   const assignGuardia = async (guardiaId: number, profesorId: number) => {
7     try {
8       // Llamada al "DAO" (servicio)
9       await updateGuardiaService(guardiaId, {
10         profesor_cubridor_id: profesorId,
11         estado: DB_CONFIG.ESTADOS_GUARDIA.ASIGNADA
12       });
13
14       // Actualización del estado
15       setGuardias(guardias.map(g =>
16         g.id === guardiaId
17         ? { ...g, profesorCubridorId: profesorId, estado: DB_CONFIG.ESTADOS_GUARDIA.ASIGNADA }
18         : g
19       ));
20
21       return true;
22     } catch (error) {
23       console.error('Error al asignar guardia:', error);
24       return false;
25     }
26   };
27
28   // Más funciones...
29
30   return (
31     <GuardiasContext.Provider value={{
32       guardias,
33       assignGuardia,
34       // Más valores y funciones
35     }}>
36       {children}
37     </GuardiasContext.Provider>
38   );
39 };
```

# Capa de presentación: Componentes React

En la capa de presentación de la aplicación utilizamos **componentes de React** para construir la interfaz de usuario. Estos componentes se encargan de renderizar las **vistas** (pantallas) y de consumir los contextos, que proporcionan datos y funcionalidades, para generar una experiencia interactiva. Esta capa es análoga a la parte de vistas en el patrón **MVC** (por ejemplo, los JSP en Java EE).

# Capa de presentación: Componentes React

```
41 // GuardiaCard.tsx
42 export const GuardiaCard: React.FC<GuardiaProps> = ({ guardia }) => {
43   const { asignarGuardia } = useGuardias();
44
45   return (
46     <div className="guardia">
47       <h3>{guardia.profesor}</h3>
48       <p>{format(guardia.fecha, 'dd/MM/yyyy')}</p>
49       {guardia.estado === 'PENDIENTE' && (
50         <Button onClick={() => asignarGuardia(guardia.id)}>
51           Asignar
52         </Button>
53       )}
54     </div>
55   );
56 };
```

# Implementación técnica: Flujo de asignación de guardia

La funcionalidad de asignar una guardia combina múltiples capas del sistema. A continuación, se representa el flujo típico cuando un profesor se asigna una guardia disponible. **Pasos en el flujo:**

- **Interfaz de Usuario:**
  - Componente React ***GuardiasPendientes.tsx*** permite al profesor ver y elegir guardias libres.
  - Al pulsar “Asignar”, se dispara un evento **handleAsignar**.
- **Contexto:**
  - Se invoca **asignarGuardia(idGuardia)** desde el ***GuardiasContext.ts***.
  - Aquí se comprueba la disponibilidad del profesor según su horario y tramos.
- **Servicio (*lib/guardiasService.ts*):**
  - **updateGuardia()** actualiza el registro en **Supabase** con el *profesorCubridorId* y cambia el estado a “ASIGNADA”.
- **Base de Datos (Supabase):**
  - La tabla guardias se actualiza con el nuevo estado y profesor asignado.
- **Actualización de UI:**
  - El contexto se actualiza y se refresca la lista de guardias del profesor.

# Enrutamiento de las páginas

El sistema de enrutamiento de **Next.js App Router** es radicalmente diferente al enfoque tradicional de Java con **servlets** o **controladores Spring**. En nuestro proyecto, cada archivo ***page.tsx*** dentro de un directorio en la carpeta ***app/*** representa automáticamente una ruta accesible.

Por ejemplo:

- ***app/admin/guardias/page.tsx*** → Accesible en ***/admin/guardias***
- ***app/profesor/ausencias/page.tsx*** → Accesible en ***/profesor/ausencias***

Este sistema de enrutamiento basado en el sistema de archivos elimina la necesidad de configuración manual de rutas, simplificando enormemente el desarrollo.

# Sistema de autenticación con Supabase y bcrypt

Para la autenticación en la aplicación, en el servicio ***authService.ts***, hemos implementado un sistema que utiliza **bcrypt** para el hash de las contraseñas y cookies para mantener las sesiones.

```
1 // Fragmento de authService.ts (implementación real)
2 import { supabase } from '../supabaseClient';
3 import bcrypt from 'bcrypt';
4
5 // Función para verificar credenciales y crear sesión
6 export async function loginUser(email: string, password: string) {
7   try {
8     // Buscar usuario por email
9     const { data: user, error } = await supabase
10       .from('usuarios')
11       .select('*')
12       .eq('email', email)
13       .single();
14
15     if (error || !user) {
16       return { error: "Usuario no encontrado" };
17     }
18
19     // Verificar contraseña con bcrypt
20     const passwordMatch = await bcrypt.compare(password, user.password);
21     if (!passwordMatch) {
22       return { error: "Contraseña incorrecta" };
23     }
24
25     // Crear sesión en cookie
26     // [Implementación específica de gestión de cookies]
27
28     return { user: { id: user.id, nombre: user.nombre, rol: user.rol } };
29   } catch (error) {
30     console.error('Error en login:', error);
31     return { error: "Error en el servidor" };
32   }
33 }
```

# Control de acceso según el rol de usuario

Para proteger las rutas según los roles de usuario:

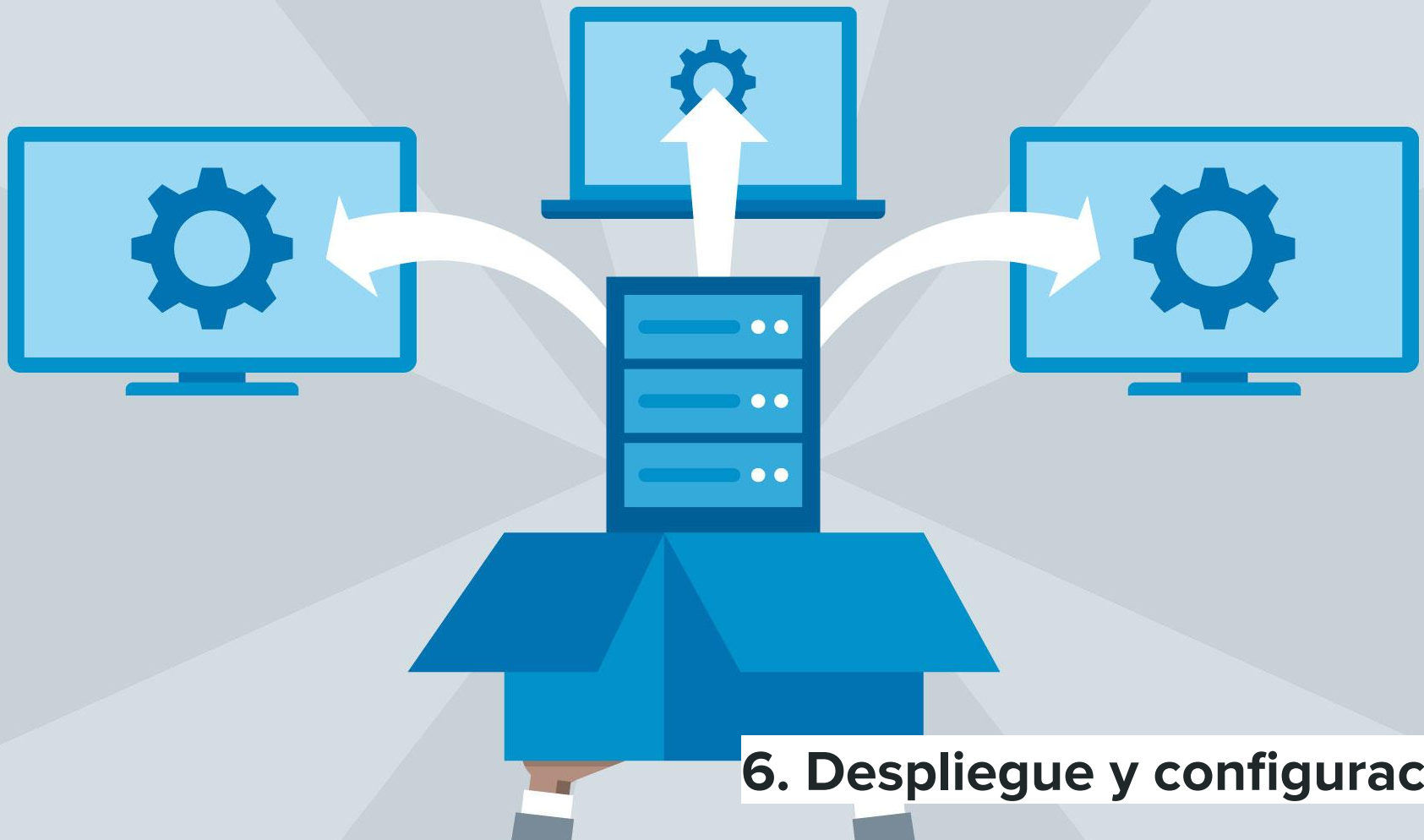
- Se utiliza el **middleware** de Next.js para interceptar las solicitudes a rutas protegidas, comprobando que el usuario tenga el rol adecuado y, en caso contrario, redirigiéndolo al login.
- Además, se aplican controles a nivel de vista para verificar el rol del usuario, lo que añade una capa extra de seguridad.

Esta doble verificación (**middleware** y **controles en la vista**) garantiza que sólo los usuarios autorizados accedan a las rutas y funcionalidades restringidas.



# Control de acceso según el rol de usuario

```
39 export function middleware(request: NextRequest) {
40   const userCookie = request.cookies.get('user')?.value
41   const isAuthPage = request.nextUrl.pathname === DB_CONFIG.RUTAS.LOGIN
42   const user = userCookie ? JSON.parse(userCookie) : null
43
44   // If no user and not on login page, redirect to login
45   if (!user && !isAuthPage) {
46     return NextResponse.redirect(new URL(DB_CONFIG.RUTAS.LOGIN, request.url))
47   }
48
49   // If user is logged in and tries to access login page
50   if (user && isAuthPage) {
51     const redirectUrl = user.rol === DB_CONFIG.ROLES.ADMIN ? DB_CONFIG.RUTAS.ADMIN : DB_CONFIG.RUTAS.PROFESOR
52     return NextResponse.redirect(new URL(redirectUrl, request.url))
53   }
54
55   // If user tries to access unauthorized routes
56   if (user) {
57     const isAdminRoute = request.nextUrl.pathname.startsWith(DB_CONFIG.RUTAS.ADMIN)
58     const isProfesorRoute = request.nextUrl.pathname.startsWith(DB_CONFIG.RUTAS.PROFESOR)
59
60     if (isAdminRoute && user.rol !== DB_CONFIG.ROLES.ADMIN) {
61       return NextResponse.redirect(new URL(DB_CONFIG.RUTAS.PROFESOR, request.url))
62     }
63
64     if (isProfesorRoute && user.rol !== DB_CONFIG.ROLES.PROFESOR) {
65       return NextResponse.redirect(new URL(DB_CONFIG.RUTAS.ADMIN, request.url))
66     }
67   }
68
69   return NextResponse.next()
70 }
71
72 export const config = {
73   matcher: [
74     '/admin/:path*',
75     '/profesor/:path*',
76     '/sala-guardias/:path*',
77     '/login'
78   ]
79 }
```



## 6. Despliegue y configuración

# Control de versiones con Git y Github

Se ha utilizado Git para controlar y gestionar el desarrollo del proyecto.

- Repositorio centralizado: <https://github.com/RGiskard7/gestion-guardias.git>
- Estructura de ramas:
  - **main:** Contiene el código estable y listo para producción, asociado al servicio de hosting.
  - **develop:** Rama de desarrollo donde se implementan nuevas funcionalidades antes de fusionarlas en main.

Tanto el código, la documentación y el vídeo de la demostración se encuentran en este repositorio.

# Configuración

Además de instalar **Node.js** y su gestor de paquetes **npm**, el proyecto requiere configurar variables de entorno específicas para conectar con **Supabase**:

- **.env.local** para desarrollo local.
- Variables configuradas en **Render** para producción.

Ejemplo de configuración:

- NEXT\_PUBLIC\_SUPABASE\_URL=https://your-project.supabase.co
- NEXT\_PUBLIC\_SUPABASE\_KEY=your-anon-key

Estas variables garantizan una conexión segura con la base de datos **PostgreSQL** y los servicios de autenticación de **Supabase**.

# Despliegue en el hosting Render

Finalmente, para facilitar el despliegue en la plataforma Render, hemos implementado:

- Archivo ***render.yaml*** que define la configuración del servicio.
- Configuración de dependencias y comandos de inicio.
- Integración con el repositorio Git para despliegue automático

Este enfoque permite un despliegue continuo donde cada cambio en la rama main se refleja automáticamente en la aplicación en producción.

# Archivo render.yaml

```
render.yaml
1  services:
2    - type: web
3      name: gestion-guardias
4      env: node
5      plan: free
6      repo: https://github.com/RGiskard7/gestion-guardias.git
7      buildCommand: npm ci && npm run build
8      startCommand: npm start
9      healthCheckPath: /
10     envVars:
11       - key: NODE_VERSION
12         value: 18.x
13       - key: NODE_ENV
14         value: production
15       - key: NEXT_PUBLIC_SUPABASE_URL
16         sync: false
17       - key: NEXT_PUBLIC_SUPABASE_ANON_KEY
18         sync: false
19       - key: SUPABASE_SERVICE_ROLE_KEY
20         sync: false
21       - key: NPM_CONFIG_PRODUCTION
22         value: false
23
```

## Enlace a la aplicación desplegada

<https://gestion-guardias.onrender.com>



## 7. Conclusiones y futuras mejoras



# Conclusiones

- **Digitalización exitosa:** La aplicación “Gestión de Guardias” ha logrado digitalizar y optimizar el proceso de gestión de guardias en centros educativos (en teoría), facilitando la administración de ausencias, asignación de guardias y seguimiento en tiempo real.
- **Arquitectura modular y escalable:** La separación de la presentación, la lógica de negocio (a través de Contextos) y el acceso a datos (con Servicios en la carpeta lib) ha permitido un desarrollo organizado y mantenible.
- **Integración con Supabase:** Utilizamos Supabase para el manejo de la base de datos PostgreSQL, lo que ha contribuido a la robustez y seguridad del sistema.
- **Seguridad reforzada:** La implementación de middleware en Next.js protege las rutas según los roles de usuario, asegurando que solo los usuarios autorizados accedan a las funcionalidades.

# Futuras mejoras

- **Notificaciones en tiempo real:** Incorporar notificaciones (por email o push) para informar a los usuarios sobre actualizaciones en guardias y ausencias.
- **Optimización de la UI/UX:** Mejorar la interfaz de usuario y adaptar la experiencia a dispositivos móviles para facilitar el uso en cualquier plataforma.
- **Ampliación de informes y estadísticas:** Desarrollar dashboards más avanzados que ofrezcan análisis detallados sobre el tiempo de guardia, distribución por tramos y otros indicadores clave.
- **Refinamiento de la gestión horaria:** Evaluar nuevas estrategias para el manejo de horarios y disponibilidades de profesores que permitan mayor flexibilidad en la asignación de guardias.
- **Integración con sistemas externos:** Explorar la posibilidad de sincronización con calendarios o sistemas de gestión educativa para optimizar la planificación.

Gracias

