ChatGPT

Plan de implementación de las interfaces de usuario del Sistema Escolar

Este documento proporciona una guía detallada para desarrollar las interfaces de usuario del proyecto **SchoolSystem**, utilizando las tecnologías específicas definidas para cada plataforma: Blazor WebAssembly/PWA (web), Flutter (móvil para padres) y .NET MAUI (aplicación de escritorio para escuelas rurales). El enfoque está en el desarrollo front-end, aprovechando la API REST y los servicios de tiempo real que ya expone el backend en .NET Core 8.0.

1. Principios generales

1. **Consistencia de experiencia de usuario**: mantener un diseño coherente entre las diferentes plataformas (web, móvil y escritorio) para que los usuarios identifiquen fácilmente secciones y funcionalidades.
2. **Separación de responsabilidades**: cada cliente consumirá la API a través de servicios encapsulados; no debe contener lógica de negocio compleja, la cual reside en el servidor.
3. **Seguridad**: todas las peticiones a la API deben ser a través de HTTPS y enviar tokens JWT en la cabecera `Authorization`. Los tokens se almacenan de forma segura (SecureStorage en móviles, ProtectedBrowserStorage en Blazor, almacenamiento cifrado en MAUI).
4. **Tiempo real**: para notificaciones y actualizaciones en vivo, se utilizará **SignalR** en todas las plataformas, suscribiéndose a los eventos que emite el servidor (nueva tarea, calificación publicada, mensajes, etc.).
5. **Soporte offline y caché local**: para entornos con conectividad limitada, se aprovecharán opciones de almacenamiento local (IndexedDB en PWA, SQLite/Hive en Flutter, SQLite en MAUI) para persistir datos esenciales y sincronizarlos cuando haya conexión.

2. Frontend Web: Blazor WebAssembly / PWA

2.1 Creación del proyecto

- Crear un nuevo proyecto Blazor WebAssembly con soporte PWA (`dotnet new blazorwasm --hosted` o configurando service worker manualmente).
- Configurar la base URL de la API en `appsettings.json` o mediante `IConfiguration` para permitir cambios según el entorno (desarrollo, producción).

2.2 Arquitectura del front

- **Componentes**: usar Razor Components para encapsular vistas (tablas de alumnos, formularios de maestros, dashboards). Separar la lógica de presentación en `*.razor.cs` cuando sea necesario.
- **Servicios Http**: crear una carpeta `Services` con clases que encapsulan llamadas a la API (GET, POST, PUT, DELETE). Cada servicio debe devolver objetos de DTO y manejar `ApiResponse<T>` y `PagedResult<T>` de la API.
- **Autenticación y autorización**: integrar la autenticación con tokens JWT mediante `Microsoft.AspNetCore.Components.Authorization`. Implementar un

`CustomAuthenticationStateProvider` que gestione inicio de sesión, almacenamiento y renovación de tokens.
- **Routing y control de acceso**: usar `@attribute [Authorize(Roles = "Admin")]` en páginas que requieren roles específicos y definir rutas en `App.razor`.
- **SignalR**: utilizar el paquete `Microsoft.AspNetCore.SignalR.Client` para conectarse al hub. Implementar un servicio singleton que gestione la conexión, reconexión y recepción de mensajes.
- **PWA y servicio offline**: configurar el service worker (`service-worker.published.js`) para cachear archivos estáticos y permitir funcionalidad offline básica. Usar IndexedDB para almacenar datos de formularios cuando no haya conexión.

3. Aplicación móvil: Flutter/Dart

3.1 Estructura inicial

- Crear un proyecto Flutter (`flutter create`) y organizarlo en módulos (features) siguiendo la arquitectura recomendada (por ejemplo, Provider o Riverpod para manejo de estado).
- Añadir paquetes necesarios: `dio` o `http` para peticiones REST, `flutter_secure_storage` para almacenar tokens, `signalr_core` para SignalR, `sqflite` o `hive` para almacenamiento local.

3.2 Integración con la API

- Implementar una clase `ApiService` para centralizar llamadas HTTP. Incluir interceptores para añadir el token JWT a las cabeceras.
- Crear modelos y servicios que representen los DTOs devueltos por la API (por ejemplo, `AlumnoDto`, `UsuarioDto`). Se puede usar `freezed` o `json_serializable` para la generación de clases a partir de JSON.
- Gestionar paginación consumiendo `PagedResult<T>` y proporcionando scroll infinito o paginación tradicional en listas.

3.3 Autenticación y roles

- Implementar una pantalla de login que capture credenciales y consuma el endpoint `/api/auth/login`. Almacenar el token de acceso mediante `flutter_secure_storage`.
- Guardar en estado global (Provider/Riverpod) la información del usuario (rol, EscuelaId) para habilitar u ocultar módulos dentro de la app según permisos.

3.4 Tiempo real con SignalR

- Usar `signalr_core` para conectarse al hub. Mantener la conexión en un servicio que pueda informar a widgets mediante Streams.
- Ejemplo: notificar a un padre cuando un maestro publique una calificación o cuando haya un mensaje nuevo en el buzón de comunicados.

3.5 Sincronización y modo offline

- Guardar datos importantes (por ejemplo, listados de calificaciones, tareas) en una base de datos local (`sqflite` o `hive`).
- Implementar un mecanismo de sincronización al detectar conectividad (podría usarse `connectivity_plus` para saber el estado de la red). Enviar cambios pendientes a la API y actualizar la base local con datos frescos del servidor.

3.6 Notificaciones push (opcional)

- Integrar Firebase Cloud Messaging (FCM) para recibir notificaciones cuando Hangfire ejecute tareas programadas (por ejemplo, recordatorios de eventos). El back-end puede enviar mensajes a FCM y SignalR simultáneamente.

4. Aplicación de escritorio: .NET MAUI

4.1 Creación y configuración

- Crear un proyecto MAUI (`dotnet new maui`) y estructurarlo en capas MVVM: Models, Views, ViewModels.
- Configurar `HttpClient` y servicios de acceso a la API, reutilizando DTOs de las bibliotecas compartidas si se distribuyen en un paquete NuGet.
- Utilizar `SecureStorage` para guardar tokens.

4.2 Funcionalidad offline y sincronización

- Implementar un repositorio local con **SQLite** (usando `Microsoft.EntityFrameworkCore.Sqlite` o `sqlite-net-pcl`). Permitir que la aplicación almacene datos localmente cuando esté sin conexión.
- Crear un servicio de sincronización similar al de Flutter: detectar conectividad, enviar registros pendientes a la API y actualizar el repositorio local.

4.3 UI adaptada a escritorio

- Aprovechar controles de MAUI (DataGrid, ListView, TabView) para mostrar información de forma clara en pantallas más grandes.
- Ajustar la navegación para utilizar ventanas y menús contextuales en lugar de tabs si corresponde.
- Integrar SignalR usando la misma librería que en Blazor/Backend (`Microsoft.AspNetCore.SignalR.Client`) para notificaciones en tiempo real.

5. Comunicación en tiempo real con SignalR

Para las tres plataformas:

1. Definir un único **Hub** en el servidor (por ejemplo, `/hubs/notifications`) que emita mensajes según eventos del dominio: creación de inscripciones, registro de asistencias, calificaciones, avisos de tareas.
2. Establecer convenciones para los nombres de métodos y tipos de datos enviados (ejemplo: `ReceiveNotification(NotificationDto)`).
3. Implementar reconexión automática y manejo de reconexiones para evitar pérdida de mensajes.
4. Suscribirse a grupos basados en roles o IDs de escuela para segmentar las notificaciones (SignalR permite `Groups.AddToGroupAsync`).

6. Consideraciones adicionales

- **Internacionalización**: aunque el proyecto está en español, es recomendable preparar la estructura para soportar múltiples idiomas (uso de `IStringLocalizer` en Blazor, `intl` en Flutter y `LocalizationResourceManager` en MAUI).

- **Pruebas**: crear pruebas unitarias y de integración para los servicios de API en cada plataforma (por ejemplo, `WidgetTester` en Flutter, `bUnit` para Blazor, `xUnit` para MAUI) y pruebas de interfaz con herramientas como Cypress o Playwright para Blazor.
- **Accesibilidad**: seguir pautas de accesibilidad (contrastes, etiquetado de campos, navegación con teclado) para garantizar que la aplicación sea usable por todos.

Conclusión

La implementación de los front ends del proyecto **SchoolSystem** requiere coordinar tres tecnologías: **Blazor WebAssembly** para una aplicación web tipo PWA, **Flutter/Dart** para una app móvil dirigida a padres y **.NET MAUI** para una app de escritorio. Cada plataforma consumirá la API REST de .NET Core 8.0, gestionará la autenticación mediante tokens JWT y se suscribirá a eventos en tiempo real con **SignalR**. Además, incorporará estrategias de almacenamiento local y sincronización para funcionar en entornos con conectividad limitada, y aplicará buenas prácticas de seguridad y experiencia de usuario.
Seguir este plan facilitará el desarrollo de interfaces coherentes y mantenibles, alineadas con la arquitectura general del proyecto y las especificaciones tecnológicas definidas.
