Documento de Buenas Prácticas y Reglas de Estilo de Código para Proyectos en React

Convenciones Generales

1. **Indentación y Espaciado:**
    - Utilizar una indentación de 2 espacios.
    - No utilizar tabulaciones para la indentación.
    - Mantener una línea en blanco al final de cada archivo.
2. **Nombres de Variables y Funciones:**
    - Utilizar camelCase para nombrar variables y funciones.
    - Utilizar nombres descriptivos y significativos.
    - Evitar nombres de una sola letra, a menos que sea un caso muy específico.
3. **Componentes:**
    - Nombrar los componentes con nombres descriptivos en mayúscula inicial.
    - Utilizar fragmentos o elementos contenedores para envolver múltiples elementos JSX.
    - Separar los componentes en archivos individuales.

Estilo de Código

Principalmente seguimos los estándares y convenciones utilizados por Microsoft en su `.editorConfig`, que se encuentra en el repositorio de GitHub de .NET Runtime, con algunas desviaciones.
Todos los estilos de código a nivel de editor están implementados en nuestro propio `.editorConfig`, que se encuentra en la raíz de cada solución de C#.

1. **Declaración de Variables:**
    - Utilizar `const` para declarar variables que no se reasignan y let para aquellas que sí lo harán.
    - Utiliza la palabra clave `var` siempre que sea posible y cuando el compilador lo admita.
    - Los campos y variables de nivel de clase deben comenzar con _ (minúscula), Por ejemplo: `_miVariableDeNivelDeClase`.
    - Los campos constantes deben estar en mayúsculas.
    - Las declaraciones condicionales de una sola línea no necesitan llaves.

```
if (entry=null) throw new ArgumentException("EntryId not found");
```

- En declaraciones condicionales de múltiples líneas, la primera llave debe estar en línea con la condición seguida de la lógica y luego una nueva línea para la llave de cierre. Usa tu criterio con esto: si sientes que la llave de apertura debe estar en una nueva línea para mejorar la legibilidad, hazlo.

```
if (colType == ItemDataType.DataTime) {
    var dt = DateUtils.ParseDateTime(val);
    return dt.HasValue ? dt.Value.Ticks: default
};
```

- Los nombres de variables, propiedades y métodos deben ser explícitos y legibles para que impliquen su intención para otros que lean el código. El uso de abreviaturas como "gen" para "generar" es bienvenido, siempre que la abreviatura tenga sentido y pueda mapearse lógicamente a su versión completa de texto.
- Evita crear variables de un solo uso. Para empezar, estás aumentando la cantidad de código para leer. También estás haciendo que un revisor o mantenedor tenga que hacer referencia a la declaración de asignación para entender qué hay realmente en tu variable de un solo uso. La excepción a esta regla es si el acto de derivar el valor de un solo uso es una declaración compleja, por ejemplo, una gran ecuación o una larga cadena de llamadas a métodos/propiedades. Está bien asignar una variable de un solo uso si lo consideras necesario.

2. **Uso de Arrow Functions:**
    - Utilizar arrow functions para definir funciones, especialmente en funciones de retorno cortas y métodos de clase.
    - Pasa TODOS los parámetros opcionales utilizando etiquetas. Esto es especialmente importante cuando los parámetros opcionales comparten el mismo tipo. El propósito principal de esto no es la legibilidad, aunque eso ayuda. En cambio, es para asegurarnos de no depender implícitamente de las posiciones de los parámetros para los parámetros opcionales.
    - Las firmas de métodos largos pueden colocarse en una nueva línea para hacerlas más legibles en pantallas más pequeñas. Una vez que superes los 5 parámetros, considera una clase DTO para pasar estos datos de entrada. Si estás trabajando en una pantalla ancha, ten en cuenta a los demás que trabajan en pantallas de portátil (es decir, anchos de 1440 píxeles).
3. **Desestructuración:**
    - Utilizar la desestructuración de objetos y arreglos cuando sea posible para extraer valores de manera más clara.
4. **Uso de Props:**
    - Documentar claramente las props esperadas por cada componente.
    - Utilizar prop-types para validar las props recibidas.

Buenas Prácticas

1. **División de Responsabilidades:**
    - Mantener los componentes lo más pequeños y especializados posible.
    - Separar la lógica del componente de su representación visual utilizando contenedores y componentes de presentación.
    - Los comentarios deben informar a otros desarrolladores sobre el diseño, suposiciones y proceso de pensamiento general detrás de un trozo de código. Los comentarios no deben simplemente repetir lo que el código está haciendo claramente.
    - Por todos los medios, separa el flujo de tu código con una línea en blanco aquí y allá, pero no dejes líneas en blanco sucesivas en tu código. Mantén el flujo de código ajustado.
    - Aprovecha las oportunidades para actualizar el código circundante a estas pautas, pero no refactorices cientos de líneas. En su lugar, mantén las actualizaciones de estilo dentro del radio de tu asignación dada.
    - Al verificar enums, usa == para comparaciones simples y .In() (o .NotIn()) cuando verifiques que una variable coincida con más de uno.
2. **Gestión del Estado:**
    - Utilizar el estado local (`useState`) para el estado que es específico de un componente.
    - Utilizar context API o librerías de gestión de estado externas (como Redux) para el estado que debe ser compartido entre múltiples componentes.
3. **Manejo de Errores:**
    - Implementar manejo de errores en componentes que realizan operaciones asíncronas o que pueden fallar.
    - Utilizar `try-catch` para capturar errores en operaciones que pueden arrojar excepciones.