1.Detalla las decisiones de tu solución que consideres pertinentes.

Tratamiento de Propuestas con Stock Insuficiente:
En el caso de que una propuesta tenga stock en el almacén pero no lo suficiente para satisfacer toda la propuesta, se optó por no mostrar ese producto en la tabla de salida. Se reconoce una incertidumbre sobre cómo manejar esta casuística, ya que el documento proporcionado no especifica claramente esta situación.

Consideración de Zonas del Almacén:

Se observó que en el archivo "Stock_unificado.json" solo existe la zona "MSR", lo que indica que no se tratarán las otras zonas ("ZAR" y "SILO"). A pesar de ello, se desarrolló la lógica considerando todas las zonas, siguiendo las instrucciones del documento.

Manejo de Tiendas Físicas y Ecommerce:

Aunque el documento especifica mostrar solo los artículos que cumplen ciertos parámetros para tiendas de ecommerce(esEcommerce = 1), se desarrolló la solución considerando tanto tiendas físicas como tiendas online.
Estructura del Código:

En la carpeta "app" se encuentran los módulos encargados de crear la lógica de la solución.

En el archivo "index.js" se llama a estos módulos para ejecutar la lógica de procesamiento.

En "**getTable.js**" se definen varios métodos:

getData: Realiza el fetch del json solicitado.

getRepartoFiltered: Filtra el archivo "Prereparto_bruto.json" según las instrucciones del documento.

getTable: Por cada artículo filtrado, llama a la función "getProductStock" para obtener el stock de ese producto, creando así un array de stocks (tabla).

getProductStock: Implementa la lógica descrita en el documento para obtener el stock de cada producto, considerando la disponibilidad en las diferentes zonas del almacén y si es tienda física o ecommerce.

En "**printTable.js**" se encuentra el método que recibe un array de objetos y mediante la manipulación del DOM, crea una tabla con los datos proporcionados.

2. Describe cómo implementarías esta solución si tuvieras que acabar mostrando el resultado en un sistema low code.

a. Consideraciones de rendimiento.

b. Requisitos que necesitarías del API

a. **Consideraciones de rendimiento**:

- **Optimización de consultas**: Utilizar consultas eficientes para filtrar y recuperar los datos necesarios.
- **Caching de datos**: Si los datos no cambian con frecuencia, implementaría un mecanismo de almacenamiento en caché para evitar consultas repetidas a la base de datos.
- **Paginación**: Implementaría paginación para limitar la cantidad de datos recuperados en cada solicitud y mejorar el rendimiento.
- **Índices de base de datos**: Asegurar que la base de datos esté indexada correctamente para optimizar las consultas de búsqueda.

b. **Requisitos del API**:

- **Endpoints para la consulta de datos**: Necesitaría endpoints en el API para obtener los datos del archivo "Prereparto_bruto.json" y "Stock_unificado.json".
- **Filtrado de datos**: Los endpoints deben admitir parámetros de filtrado avanzado para permitir búsquedas precisas y flexibles. Esto puede incluir opciones para filtrar por múltiples campos, rangos de valores y operadores lógicos.
- **Documentación clara**: El API debe tener una documentación clara que describa cómo acceder a los endpoints, los parámetros admitidos y el formato de los datos de respuesta.
- **Seguridad**: Debería haber medidas de seguridad implementadas en el API para proteger los datos confidenciales y garantizar que solo los usuarios autorizados puedan acceder a ellos.
- **Escalabilidad**: El API debe ser escalable para manejar un aumento en la carga de solicitudes, especialmente si se espera un crecimiento en el volumen de datos o en el número de usuarios del sistema low code.

3. Si el json real del prerepato ocupase 20Gb, explica si el problema de forma distinta y por qué.

1. Utilizaría **Node.js** para aprovechar su funcionalidad de lectura de archivos en **streams**. Esto permite procesar grandes archivos de manera eficiente y sin consumir demasiada memoria.
2. Dividir el archivo en **lotes** o chunks para facilitar la transmisión y la gestión de datos. Esto se logra leyendo el archivo en bloques más pequeños y enviando cada lote de datos por separado.
3. Establecer un endpoint específico, por ejemplo, "/reparto", que acepte un parámetro de consulta "lote" para solicitar un lote específico del archivo. Por ejemplo, "/reparto?lote=1" recupera el primer lote de datos.
4. Cuando se solicita un lote específico, el servidor responderá con un objeto JSON que contiene el lote de datos en formato de cadena, el número del lote y un indicador de si es el último lote. Por ejemplo:

```
{
  "lote": "...", // el JSON en formato string
  "num_lote": number, // el número de lote
  "last_lote": boolean// indica si es el último lote
}
```

Frontend:

1. En el frontend, utilizar el método **`fetch()`** para enviar solicitudes al endpoint "/reparto" con el parámetro de consulta "lote" correspondiente al número de lote que se desea recuperar. Y gracias a last_lote implementaría una función recursiva para comprobar si ha llegado o no al último lote:

```
solicitarLotes (loteActual):

    respuesta = await fetch(`/reparto?lote=${loteActual}`);
    datos = await respuesta.json();

    // Procesar el lote de datos recibido
            Const  table =await  getTable(datos.lote);
            printTable(tabla)

    // Verificar si es el último lote
    if(datos.last_lote):
        // Si no es el último lote, solicitar el siguiente lote recursivamente
       await solicitarLotes(data.num_lote + 1);
```

```
else:
    // ultimo lote
```

4. Si tuvieras que de forma visual presentar en una pantalla desde que partes de un almacén se rellena un pedido, que propuesta de visualización plantearías teniendo en cuenta que se quiere implementar con una herramienta low code.

- **Mapa interactivo del almacén**:
    - Crearía un mapa interactivo del almacén que muestre las diferentes zonas de almacenamiento, como ZAR, MSR y SILO.
    - Cada zona del almacén estaría representada por un área en el mapa, con etiquetas que indiquen el tipo de stock almacenado en cada zona.
    - Al seleccionar una zona del almacén, se mostrarían detalles adicionales, como los artículos disponibles, la cantidad de stock y la disponibilidad para satisfacer el pedido.
- **Visualización de pedido:**
    - Mostrar el pedido actual que se está procesando en la pantalla.
    - Indicar visualmente los productos del pedido en el diagrama del almacén.
    - Resaltar las ubicaciones de los productos que se están utilizando para satisfacer el pedido.
- **Seguimiento en tiempo real:**
    - Actualizar la visualización en tiempo real a medida que se van utilizando los productos del almacén para satisfacer el pedido.
    - Mostrar animaciones o efectos visuales para indicar cómo se mueven los productos desde el almacén hasta el área de preparación del pedido.
- **Información adicional:**
    - Proporcionar información adicional sobre el estado del pedido, como el progreso general, el tiempo estimado de preparación, etc.
    - Permitir al usuario interactuar con la visualización, como hacer zoom para ver detalles específicos del almacén o hacer clic en los productos para obtener más información.
- **Diseño intuitivo y fácil de entender:**
    - Mantener el diseño limpio y simple para que sea fácil de entender incluso para usuarios no técnicos.
    - Utilizar etiquetas claras y descriptivas para identificar cada parte del almacén y cada producto en el pedido.