PROPUESTA LIBRERIA @google-cloud/storage

Librería Oficial: @google-cloud/storage

Es la opción estándar y más robusta. Permite manejar **Streams**, lo cual es vital: en lugar de cargar los 12GB en la RAM, el archivo se lee en pequeños pedazos (chunks) y se envía secuencialmente.

Cómo aplicarlo (Stream dinámico):

JavaScript

```
const { Storage } = require('@google-cloud/storage');
const fs = require('fs');

const storage = new Storage();
const bucket = storage.bucket('tu-bucket-grande');

async function uploadLargeFile() {
  const file = bucket.file('archivo-gigante.zip');

  // Crea un stream de lectura del archivo local
  fs.createReadStream('./local-12gb-file.zip')
    .pipe(file.createWriteStream({
      resumable: true, // Habilita la subida reanudable automáticamente
      contentType: 'application/zip',
      validation: 'crc32c' // Verifica la integridad del archivo masivo
    }))
    .on('error', (err) => console.error('Error:', err))
    .on('finish', () => console.log('Subida completada con éxito.'));
}
```

Es la librería nativa de Google. Está diseñada para integrarse perfectamente con los servicios de GCP y soporta de forma nativa los **Resumable Uploads**.

Ventajas:

Nativa: Soporte total para autenticación de GCP (Service Accounts, ADC).

Resumable por defecto: Maneja automáticamente la creación de la sesión de subida reanudable.

Integridad: Incluye validación de hashes (CRC32C) para asegurar que los 12GB llegaron íntegros.

Desventajas:

Acoplamiento: Te amarra al ecosistema de Google. Si decides cambiar de nube, debes reescribir la lógica.

Configuración Manual: Para optimizar la velocidad en archivos masivos (paralelismo), requiere configuración manual de chunks.

Ideal para: Aplicaciones que corren dentro de GCP (Cloud Run, GKE) y necesitan máxima compatibilidad.

Enfoque Hibrido?

Para un caso de **Arquitectura de Sistemas de Alta Disponibilidad**:

Backend (Node.js): Usa la **Librería Oficial @google-cloud/storage** para generar **Signed URLs Resumable**. Esto permite que el cliente suba directamente al bucket sin pasar por tu servidor (evitando que tu RAM colapse con 12GB).

Frontend: Usa **tus-js-client** o simplemente lógica de **fetch/axios** apuntando a la Signed URL de Google.

¿Por qué este combo? Porque delegas la carga pesada a la infraestructura global de Google (Alta Disponibilidad garantizada) y usas una Signed URL para mantener la seguridad. Google Storage ya implementa su propia versión de "resumable uploads" que es tan eficiente como TUS pero sin necesidad de instalar servidores adicionales como

¿Cómo lo hace Google Cloud Storage?

GCS utiliza un protocolo propio basado en HTTP. Cuando inicias una subida reanudable, Google te entrega una **Session URI** única.

Si la conexión se corta, el cliente hace una petición PUT vacía a esa misma URI para preguntar: *"¿Cuántos bytes tienes ya?"*.

Google responde con el rango (ej. 0-5000000).

El cliente reanuda el envío desde el byte 5000001.

Usar tus-js-client con Google Cloud Storage

puedes usar la librería de TUS para subir a Google Cloud.
¿Por qué harías esto? Porque la librería tus-js-client es mucho más inteligente gestionando reintentos automáticos, retardos exponenciales (exponential backoff) y el estado en el localStorage del navegador que un simple script de fetch.

Ejemplo de cómo se ve el flujo de "Reanudación" en Node.js (Oficial)

Si tu servidor de Node.js se cae a mitad de una subida de 12GB, puedes recuperar la sesión así:
JavaScript

```
const file = bucket.file('archivo-grande.zip');

// Imagina que guardaste esta URI en una base de datos cuando empezó la subida
const uploadSessionUri =
'https://storage.googleapis.com/upload/storage/v1/b/bucket/o/archivo...';

const writeStream = file.createWriteStream({
  uri: uploadSessionUri, // Reanuda usando la sesión existente
  resumable: true
});

fs.createReadStream('./localfile.zip',{start:bytesYaSubidos}).pipe(writeStream
);
```

Para implementar una arquitectura de alta disponibilidad para archivos de **+12GB** usando la librería oficial de @google-cloud/storage, lo ideal es el patrón de **Subida Reanudable (Resumable Upload)**.
Aquí tienes el paso a paso dividido en el **Backend** (quien autoriza) y el **Frontend** (quien sube los bytes).

Paso 1: Configuración del Proyecto y Bucket

Instalar la librería:
Bash
```
npm install @google-cloud/storage
```

CORS: Es vital configurar el bucket para permitir que el navegador envíe archivos grandes. Crea un archivo cors-json.json:
JSON
```
[
  {
    "origin": ["*"],
    "method": ["PUT", "POST"],
    "responseHeader": ["Content-Type", "x-goog-resumable"],
    "maxAgeSeconds": 3600
  }
]
```
Y aplícalo con gcloud storage buckets update gs://TU_BUCKET --cors-file=cors-json.json.

Paso 2: Backend (Node.js) - Generar la Sesión de Subida

No queremos que los 12GB pasen por tu servidor. Tu servidor solo pedirá a Google una **"Session URI"** y se la dará al cliente.
JavaScript
```
const { Storage } = require('@google-cloud/storage');
const storage = new Storage(); // Asegúrate de tener las credenciales en el
entorno

async function getResumableUrl(fileName, fileType) {
  const bucket = storage.bucket('tu-bucket-grande');
  const file = bucket.file(fileName);

  // Generamos una Signed URL configurada para subidas reanudables
  const [url] = await file.getSignedUrl({
    version: 'v4',
    action: 'write',
    expires: Date.now() + 60 * 60 * 1000, // 1 hora para iniciar
    contentType: fileType,
    extensionHeaders: {
      'x-goog-resumable': 'start', // Esto indica que inicia una sesión
reanudable
    },
  });

  return url;
}
```
Al llamar a esta URL (haciendo un POST), Google devolverá en los headers una location que es la **Session URI** definitiva para subir los bytes.

Paso 3: Frontend (JavaScript) - Subida por Partes (Chunks)

Para archivos de 12GB, no puedes hacer un solo PUT. Debes dividirlo en trozos (chunks) para que, si falla uno, solo reintentas ese trozo.

JavaScript

```
async function uploadLargeFile(file, sessionUri) {
  const CHUNK_SIZE = 32 * 1024 * 1024; // 32MB por trozo
  let start = 0;

  while (start < file.size) {
    const end = Math.min(start + CHUNK_SIZE, file.size);
    const chunk = file.slice(start, end);

    const response = await fetch(sessionUri, {
      method: 'PUT',
      headers: {
        'Content-Range': `bytes ${start}-${end - 1}/${file.size}`,
      },
      body: chunk,
    });

    if (response.status === 308) {
      // 308 Resume Incomplete: Google recibió el trozo, seguimos
      start = end;
      console.log(`Progreso: ${Math.round((start / file.size) * 100)}%`);
    } else if (response.status === 200 || response.status === 201) {
      console.log('¡Subida completa!');
      break;
    } else {
      throw new Error('Error en la subida');
    }
  }
}
```

En el Backend (Node.js) con @google-cloud/storage

Si estás subiendo el archivo desde un servidor (por ejemplo, procesando un archivo que ya está en el disco de tu server), la librería **ya maneja el chunking por defecto**.
Cuando haces esto:
JavaScript

```
file.createWriteStream({ resumable: true })
```

La librería crea un buffer interno. A medida que le pasas datos (mediante .pipe() o .write()), ella espera a juntar una cantidad de bytes (por defecto 16MB) y los envía como un chunk a GCS. Si la conexión falla, la librería intenta reanudar desde el último chunk confirmado.

2. En el Frontend (Navegador): La lógica manual

Aquí es donde debes tener cuidado. El navegador no tiene una función nativa de "subida reanudable automática" para archivos de 12GB. Si intentas enviar los 12GB en un solo fetch, lo más probable es que la conexión muera o el navegador se bloquee.
¿Cómo se aplica la lógica de chunking manual? Debes usar el método .slice() del objeto File (que es un Blob). Este método **no carga el archivo en la memoria RAM**, sino que crea una referencia a una parte del archivo en el disco.

El Algoritmo de Chunking:

Dividir: Defines un tamaño de trozo (ej. 32MB). Google exige que sea múltiplo de 256KB.

Iterar: Usas un bucle para recorrer el archivo desde el byte 0 hasta el final.

Enviar con Header Content-Range: Este es el paso más importante. Debes decirle a Google qué pedazo estás enviando.

Ejemplo de implementación real:

JavaScript

```
const CHUNK_SIZE = 32 * 1024 * 1024; // 32MB (óptimo para GCS)

async function uploadInChunks(file, sessionUri) {
  let start = 0;
  const totalSize = file.size;

  while (start < totalSize) {
    const end = Math.min(start + CHUNK_SIZE, totalSize);
    const chunk = file.slice(start, end); // Esto es instantáneo y no gasta
RAM

    try {
      const response = await fetch(sessionUri, {
        method: 'PUT',
        headers: {
          // Formato: bytes inicio-fin/total
          'Content-Range': `bytes ${start}-${end - 1}/${totalSize}`,
        },
        body: chunk,
      });

      if (response.status === 308) {
        // 308 Resume Incomplete: Google dice "lo recibí, manda el que sigue"
        start = end;
      } else if (response.ok) {
        console.log("Subida terminada");
        break;
      }
    } catch (error) {
      console.error("Conexión perdida. Reintentando en 5 segundos...");
      await new Promise(r => setTimeout(r, 5000));
      // Aquí podrías llamar a una función para preguntar a Google
      // en qué byte se quedó antes de seguir el bucle.
    }
  }
}
```

@Google-cloud/storage y tus-js-client

El truco aquí es que **tus-js-client** es extremadamente inteligente para manejar reintentos y estados locales, mientras que **GCS** tiene un protocolo resumable que, aunque no es TUS puro, se puede "engañar" o adaptar fácilmente.
Dado que GCS no soporta el protocolo TUS de forma nativa sin un servidor intermedio (como tusd), la forma más limpia de hacerlo sin añadir más servidores es usar **tus-js-client** configurado para hablar con el endpoint resumable de Google.
Aquí tienes el paso a paso:

1. Backend (Node.js): Generar la Session URI

El backend solo se encarga de pedirle a Google una "sesión de subida". Esta sesión es una URL especial que tus-js-client usará para enviar los bytes.

JavaScript

```
const { Storage } = require('@google-cloud/storage');
const storage = new Storage();

// Endpoint que llamará tu frontend antes de empezar la subida
async function createUploadSession(req, res) {
  const bucket = storage.bucket('tu-bucket');
  const file = bucket.file(req.body.fileName);

  // 1. Iniciamos una sesión resumable
  // Esto no sube el archivo, solo crea el "túnel" en GCS
  const [url] = await file.getSignedUrl({
    version: 'v4',
    action: 'write',
    expires: Date.now() + 3600 * 1000, // 1 hora
    contentType: req.body.fileType,
    extensionHeaders: {
      'x-goog-resumable': 'start',
    },
  });

  // 2. IMPORTANTE: GCS requiere un POST inicial a esa URL para darte la
SessionID
  // Pero podemos delegar eso al cliente o hacerlo aquí.
  // Lo más estable es enviarle la Signed URL al cliente.
  res.json({ uploadUrl: url });
}
```

---

2. Frontend: Configurar tus-js-client

Normalmente, TUS espera un servidor TUS, pero puedes usar el **CustomHttpClient** o simplemente adaptar las opciones para que coincidan con lo que GCS espera (especialmente los headers de rango).

Sin embargo, para archivos de 12GB en GCS, hay una forma más directa: **TUS tiene un adaptador para el protocolo de Google**.

Instalación

Bash

```
npm install tus-js-client
```

Implementación

JavaScript

```
import * as tus from 'tus-js-client';

async function startUpload(file) {
  // 1. Obtener la URL firmada de tu backend
  const response = await fetch('/api/get-signed-url', {
    method: 'POST',
    body: JSON.stringify({ fileName: file.name, fileType: file.type })
  });
```

```
  const { uploadUrl } = await response.json();

  // 2. Configurar TUS
  const upload = new tus.Upload(file, {
    // GCS usa su propio endpoint, le pasamos la URL que generó el backend
    uploadUrl: uploadUrl,
    endpoint: uploadUrl,

    // Configuraciones de resiliencia para 12GB
    retryDelays: [0, 3000, 5000, 10000, 20000], // Reintentos automáticos si
cae el Wi-Fi
    chunkSize: 32 * 1024 * 1024, // 32MB por bloque

    metadata: {
      filename: file.name,
      filetype: file.type
    },

    // Adaptar headers para Google Cloud Storage
    onBeforeChunkUpload: (id, chunkSize, offset) => {
      return {
        addHeaders: {
          // GCS usa Content-Range para subidas resumables
          'Content-Range': `bytes ${offset}-${offset + chunkSize -
1}/${file.size}`
        }
      };
    },

    onError: (error) => {
      console.error("Fallo crítico:", error);
    },

    onProgress: (bytesSent, bytesTotal) => {
      const percentage = (bytesSent / bytesTotal * 100).toFixed(2);
      console.log(`Progreso: ${percentage}%`);
    },

    onSuccess: () => {
      console.log("¡Archivo de 12GB subido con éxito a GCS!");
    }
  });

  // Iniciar subida
  upload.start();
}
```