Vamos agora reescrever o mesmo exercício da última prática sugerida utilizando *async/await*, uma forma mais moderna e legível de lidar com código assíncrono em *JavaScript*.

JS asyncawait.js

```
function baixarConteudoPromise(nomeConteudo) {
    return new Promise((resolve, reject) => {
      console.log(`Iniciando o download de ${nomeConteudo}...`);

      // Simula um tempo de download (em milissegundos)
      setTimeout(() => {
        const sucesso = true; // Simula se o download foi bem-sucedido

        if (sucesso) {
          console.log(`${nomeConteudo} foi baixado com sucesso.`);
          resolve(nomeConteudo); // Resolvendo a Promise com sucesso
        } else {
          const erro = 'Erro ao baixar o conteúdo';
          console.error(erro);
          reject(erro); // Rejeitando a Promise em caso de erro
        }
      }, 2000); // Simula 2 segundos de download
    });
  }

  async function baixarConteudoAsyncAwait() {
    try {
      const documento = await baixarConteudoPromise('Documento.pdf');
      console.log(`Download de ${documento} concluído com sucesso!`);

      const imagem = await baixarConteudoPromise('Imagem.jpg');
      console.log(`Download de ${imagem} concluído com sucesso!`);

      const video = await baixarConteudoPromise('Video.mp4');
      console.log(`Download de ${video} concluído com sucesso!`);
    } catch (erro) {
      console.error(`Erro durante o download: ${erro}`);
    }
  }

  // Chama a função assíncrona
  baixarConteudoAsyncAwait();
```

Neste exemplo, utilizamos *async* na declaração da função baixarConteudoAsyncAwait, permitindo o uso de *await* dentro dela para lidar com *Promises* de forma síncrona. Isso torna o código mais limpo e fácil de entender.

O *try* e *catch* são usados para lidar com erros, assim como no exemplo anterior com *Promises*. O código dentro do bloco *try* aguarda a resolução de cada *Promise* antes de prosseguir para o próximo passo. Se algum erro ocorrer, o bloco *catch* captura o erro e trata conforme necessário.